J.S. van der Laan

Optimal Routing Algorithms

A flow-based approach and speed-up techniques for static networks

Master's thesis

Supervisor: Dr. F.M. Spieksma

Date of exam: 29 November 2017

Mathematisch Instituut, Universiteit Leiden

Optimal Routing Algorithms

A flow-based approach and speed-up techniques for static networks

Student name:	J.S. van der Laan
Student ID:	s1281321
Master program:	Mathematics
Specialisation:	Applied Mathematics
Department:	Mathematisch Instituut, Universiteit Leiden
Date:	November 23, 2017
Supervisor:	Dr. F.M. Spieksma

Abstract

Many practical optimisation problems can be formulated as a traffic assignment problem, i.e. optimally route a multi-commodity flow through a network. In order to do this, a network is defined that can capture congestion and a notion of optimal flow. The shortest path problem is derived as a sub-problem of the traffic assignment problem, discussing several algorithms that can solve it. In addition, several speed-up techniques for the shortest path problem are described that can be applied to static networks. In conclusion, an algorithm is discussed that solves the traffic assignment problem by iteratively solving a shortest path problem.

Contents

Introduction

Routing a flow optimally through a network represents a remarkable amount of practical problems, occurring in fields such as transportation, computer science, operations research, chemistry and social science. Graphs are often used as the mathematical representation of the networks to capture the abstract structure of such optimisation problems. An impressive amount of research has been done on graphs and optimisation related problems, resulting in numerous algorithms that solve the optimal routing problem. This thesis contains an assembly of some interesting algorithms that depicts the variety in approaches and the mathematical elegance that intertwines them.

Chapter 1 states the necessary notation and definitions to create the abstract framework that is used throughout this thesis. The definition of a multi-commodity flow and a notion of optimality leads to the routing problem that is referred to as the traffic assignment problem.

Chapter 2 derives the shortest path problem as a sub-problem of the traffic assignment problem. Other research on the shortest path problem does, in general, not have a flow-based framework. In this thesis, the flow-based framework is chosen deliberately to emphasise the similarity of the two problems and will lead directly to a linear program that is then solved by the classic Primal-Dual Algorithm. Furthermore, a generic labelling algorithm is given, from which the two well-known Bellman-Ford and (Bidirectional) Dijkstra algorithms are obtained.

When networks are static, in the sense that many shortest path queries are made without changing the network, information can be retrieved in advance. This information can then be used to improve the query-times. Several such speed-up techniques are discussed in Chapter 3, including a surprising return of the Primal-Dual Algorithm and a dominant use of the Bidirectional Dijkstra algorithm.

A second notion of optimality will be introduced in Chapter 4, together with a proof of equivalence concerning the first notion of optimality. In conclusion, the optimal routing problem, with respect to the second notion, is solved by the Frank-Wolfe algorithm. Applying the Frank-Wolfe algorithm shows that the general routing problem can be solved by an iterative process of shortest path computations, once again emphasising the mathematical elegance of the subject.

1. Preliminaries

This chapter will introduce the notation and definitions that will be used.

Let $G = (V, A)$ be a directed graph with $|V| = n < \infty$ vertices and $|A| = m < \infty$ edges, where $a_{u,v} := (u, v) \in A$ is a directed edge from $u \in V$ to $v \in V$. The considered graphs are non-empty, connected and without multiple edges with the same direction between two nodes.

Define a set of *source nodes*, $S \subseteq V$, and a set of *target nodes*, $T \subseteq V$. The pairs $(s_1, t_1), \ldots, (s_k, t_k)$, for $s_i \in S$ and $t_i \in T$, are called *origin-destination pairs (O-D pairs)*. The set of O-D pairs does not necessarily contain all possible s-t pairs, with $s \in S$ and $t \in T$. Every O-D pair (s_i, t_i), for $i \in \{1, \ldots, k\}$, has a set of possible paths from s_i to t_i, denoted by $\mathcal{P}_i$.

Assumption 1.0.1. We only consider O-D pairs for which $\mathcal{P}_i \neq \emptyset$. In other words, there always exists at least one path that connects s_i to t_i, for every O-D pair (s_i, t_i), with $i \in \{1, \ldots, k\}$.

Furthermore, define $\mathcal{P} := \bigcup_{i=1}^{k} \mathcal{P}_i$. To each path $P \in \mathcal{P}$ we associate a path flow $f_P \in [0, \infty)$. The flow f on a graph is the collection of all path flows f_P, for $P \in \mathcal{P}$, i.e. $f = \bigcup_{P \in \mathcal{P}} f_P$. The flow on a single edge $a \in A$ is given by

$$f_a = \sum_{P \in \mathcal{P}} \delta_P^a f_P, \quad \text{where } \delta_P^a := \begin{cases} 1, & \text{if } a \in P \\ 0, & \text{otherwise} \end{cases}. \tag{1.0.1}$$

Note that f_a is the amount of flow between all O-D pairs that passes through the edge a, where f_P only represents the flow on one path, for one O-D pair. Also note that a flow f does not necessarily satisfy the usual flow conservation property. See for example Figure 1.2.

Between each s_i-t_i pair, a prescribed amount of flow has to be sent, denoted by the *flow rate* (or *demand*) r_i. The triple $C_i := (s_i, t_i, r_i)$ is referred to as a *commodity*, where $\mathcal{C} := \{C_1, \ldots, C_k\}$ is the set of all commodities.

Definition 1.0.2 (Feasible flow)**.** A flow f is called *feasible* if, for each commodity $C_i \in \mathcal{C}$, with $i \in \{1, \ldots, k\}$, the following properties hold.

- $\sum_{P \in \mathcal{P}_i} f_P = r_i$,
- $f_P \geq 0$, for all $P \in \mathcal{P}_i$.

Furthermore, every edge $(u, v) = a \in A$ is assigned a non-negative, continuous and non-decreasing weight function $c_a : \mathbb{R}^+ \to \mathbb{R}^+$, also denoted by $c_{u,v}$, mapping the amount of flow on the edge to the corresponding weight. This weight function is an abstract quantity that can represent many features of an edge. The function is non-decreasing to capture congestion on edges with higher flow throughput. For a feasible flow f, the total weight of a path P is given by

$$c_P(f) := \sum_{a \in P} c_a(f_a). \tag{1.0.2}$$

The total weight of the flow f is given by

$$z(f) := \sum_{P \in \mathcal{P}} c_P(f) f_P = \sum_{a \in A} c_a(f_a) f_a. \tag{1.0.3}$$

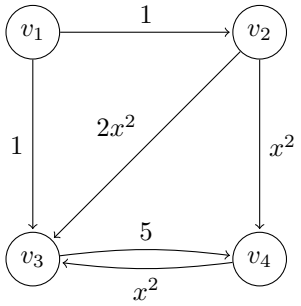


Figure 1.1: A graph G with edge weight functions stated besides the edges.

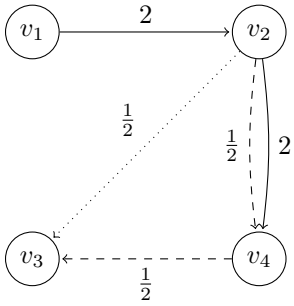


Figure 1.2: A feasible flow f for the graph G, given in Figure 1.1.

A graph G, together with a set of commodities $\mathcal{C}$ and a weight function c, as described above, define an instance $G_{\mathcal{C},c}$. Note that such an instance can have many different feasible flows.

The following is an example of an instance with assigned flow.

Example 1.0.3. Consider the graph given in Figure 1.1 together with the commodities $C_1 := (v_1, v_4, 2)$ and $C_2 := (v_2, v_3, 1)$. The two O-D pairs $(s_1, t_1) := (v_1, v_4)$ and $(s_2, t_2) := (v_2, v_3)$ have path sets

$$\begin{aligned} \mathcal{P}_1 &:= \{(v_1, v_3, v_4), (v_1, v_2, v_4), (v_1, v_2, v_3, v_4)\} \text{ and} \\ \mathcal{P}_2 &:= \{(v_2, v_3), (v_2, v_4, v_3)\} \end{aligned}$$

respectively.

A possible feasible flow f is given in Figure 1.2. For commodity C_1, the flow is sent along only one of the three possible paths. For commodity C_2, the flow is sent along all possible paths. This results in the following flow.

$$f_{(v_1,v_2,v_4)} = 2, \quad f_{(v_1,v_3,v_4)} = 0, \quad f_{(v_1,v_2,v_3,v_4)} = 0, \tag{1.0.4}$$

$$f_{(v_2,v_3)} = \tfrac{1}{2} \quad \text{and} \quad f_{(v_2,v_4,v_3)} = \tfrac{1}{2}. \tag{1.0.5}$$

Note that the flow on edge (v_2, v_4) is the sum of the flows along all paths, so $f_{a_{v_2,v_4}} = 2\frac{1}{2}$.

Some thought reveals that changing the flow f to another feasible flow in G will result in a higher total weight $z(f)$. So, the flow f in Figure 1.2 is the flow that minimises the total weight. This leads to the following definition.

Definition 1.0.4 (System optimum)**.** Let $G_{\mathcal{C},c}$ be an instance. Then a flow f is called the *system optimum (SO)* if it solves the problem

$$\begin{aligned} \text{minimize} \quad & z(f) \\ \text{subject to} \quad & \sum_{P \in \mathcal{P}_i} f_P = r_i, \quad i \in \{1, \dots, k\} \\ & f_P \geq 0, \qquad\qquad P \in \mathcal{P} \end{aligned} \quad . \tag{1.0.6}$$

Finding such a flow is referred to as the *traffic assignment problem*.

Note that, although in practice often the case, the edges do not have a capacity constraint. However, one could incorporate a capacity constraint by assigning a weight function to an edge that is asymptotic at the desired capacity. Minimising program (1.0.6) will then prevent the flow from exceeding the capacity.

2. Shortest Paths

2.1 Problem definition

In this chapter we will formulate the shortest path problem using the notation introduced in Chapter 1. Furthermore, several algorithms will be discussed that can solve this problem.

We will formulate the shortest path problem as the problem of finding the SO flow for a special instance $G_{\mathcal{C},c}$. These special instances will be defined as follows. Let S be a set of source nodes and T be a set of target nodes, with the desired O-D pairs $(s_1, t_1), \ldots, (s_k, t_k)$, for which the shortest path must be found. Let $r_i = 1$, for all $i \in \{1, ..., k\}$, and let c_a be a constant function[1] of the flow, for all $a \in A$. This way, only one unit of flow has to be sent between the source and target node of the O-D pairs, without the effect of congestion. Note that for O-D pair (s_i, t_i) the unit of flow could, in general, be split between more than one path. However, it is intuitively clear that all flow will be sent along the path with minimal weight, and thus the shortest path between s_i and t_i. The weight of such a path is defined as the *distance* from s_i to t_i, i.e.

$$d_{s_i,t_i} := \min_{P \in \mathcal{P}_i} c_P. \tag{2.1.1}$$

So, a shortest path problem comes down to finding the flow that solves (1.0.6) for an instance as described above. This will implicitly give the shortest paths between all O-D pairs.

Three main sub problems can be distinguished.

- $S = V = T$ – All possible pairs of nodes are an O-D pair. This is referred to as the *all-pairs shortest path (APSP)* problem.
- $|S| = 1$ – This is referred to as a *single source shortest path (SSSP)* problem. When considering an SSSP problem, we will always denote $S := \{s\}$.
- $|S| = 1 = |T|$ – This is referred to as a *point-to-point (P2P)* shortest path problem. When considering a P2P problem, we will always denote $S := \{s\}$ and $T := \{t\}$.

2.2 Labelling Algorithms

Two well-known algorithms that solve the SSSP problem (as well as the P2P problem) are the Bellman-Ford algorithm and Dijkstra's algorithm. These two algorithms can both be seen as special cases of one generic labelling algorithm.

The labelling algorithm labels every node $v \in V$ with the following two values:

- d_v^* – The tentative distance from s to v.
- $pred(v)$ – The predecessor node of v on the shortest path from s to v.

[1] While considering a shortest path problem, the weight of an edge will be referred to as c_a, instead of $c_a(f_a)$, since it is constant.

These values will be updated until $d^*_{t_i} = d_{s,t_i}$, for all O-D pairs (s, t_i).

Initially s has tentative distance zero and no predecessor. For every other node the tentative distance is set to ∞, also with no predecessor. During the algorithm an edge $(u, v) \in A$ is called *tense* if

$$d^*_v > d^*_u + c_{u,v}. \tag{2.2.1}$$

Let $(u, v) \in A$ be tense. The stored tentative distance to v is incorrect, since d^*_v can be improved. The edge can be *relaxed* by updating the tentative distance of v, in other words by putting

$$d^*_v \leftarrow d^*_u + c_{u,v}. \tag{2.2.2}$$

If there are no tense edges in the graph left, a shortest path tree from s is found and the algorithm stops, since, for a shortest s-t path P, all edges $(u, v) \in P$ have

$$d^*_v = d_{s,v} = d^*_u + c_{u,v}. \tag{2.2.3}$$

Remark 2.2.1. For all $v \in V$ and at arbitrary time during the algorithm, we have

$$d^*_v \geq d_{s,v} \tag{2.2.4}$$

The order in which the edges of the graph are relaxed is imposed by a `Box` of vertices, which initially only contains source node s. A vertex is pulled from the `Box` and all its outgoing tense edges are relaxed. After that, the vertex at the end of a relaxed edge is put into the `Box`. Once the `Box` is empty, there are no more tense edges and the shortest path tree is found. See Algorithm 1.

Algorithm 1 Generic Labelling algorithm

```
1:  procedure GenericLabelling(s)
2:      Init(s);
3:      Box ← {s};
4:      while Box ≠ ∅ do
5:          pull u from the Box;
6:          for all (u, v) ∈ A do
7:              if (u, v) tense then
8:                  Relax(u, v);
9:                  Box ← Box ∪ {v};
10:             end if
11:         end for
12:     end while
13: end procedure

function Init(s)
    d*_s ← 0;
    pred(s) ← ∅;
    for s ≠ v ∈ V do
        d*_v ← ∞;
        pred(v) ← ∅;
    end for
end function

function Relax(u, v)
    d*_v ← d*_u + c_{u,v};
    pred(v) ← u;
end function
```

Note that the way vertices are pulled from the `Box` (Line 5 of Algorithm 1) and stored inside the `Box` (Line 9 of Algorithm 1) is not made explicit. The Bellman-Ford algorithm and Dijkstra's algorithm use different implementations of the `Box`, resulting in two algorithms that, at first glance, may occur as unrelated.

2.2.1 The Bellman-Ford algorithm [2, 10]

In the case of the Bellman-Ford algorithm, the `Box` is a queue with the first-in-first-out (FIFO) property. After scanning[2] vertex s, all shortest paths of length one are found. After scanning all vertices that were queued while scanning s, all shortest paths of length at most two are found and after $|V| - 1$ iterations the algorithm has found all shortest paths of length at most $|V| - 1$. Lemma 2.2.2 shows that there always exists a shortest path with at most $|V| - 1$ edges. Hence, after $|V| - 1$ times scanning all queued vertices, a shortest path from s to every other nodes is found.

Moreover, $|V| - 1$ times scanning all queued vertices is equivalent to $|V| - 1$ times relaxing all tense edges in the graph. This results in the algorithm that is often referred to as the Bellman-Ford(-Moore) Algorithm. See Algorithm 2. The time complexity of this algorithm is, straightforwardly, $O(nm)$.

Algorithm 2 The Bellman-Ford algorithm

```
1: procedure BellmanFord(s)
2:     Init(s);
3:     repeat |V| − 1 times
4:         for all a ∈ A do
5:             if a tense then
6:                 Relax(a);
7:             end if
8:         end for
9:     end repeat
10: end procedure
```

Lemma 2.2.2. *Let $G = (V, A)$ be a directed graph with non-negative edge weights c_a. There exists a shortest path of at most $|V| - 1$ edges, between every two nodes $u, v \in V$.*

Proof. Let $P = (u = p_1, p_2, \ldots, p_{\ell+1} = v)$ be a shortest path with ℓ edges. If no node is visited more than once, we have $\ell + 1 \leq |V|$ and we are done. Suppose p^* is a node that is visited more than once. Let P^* be the subpath, from p^* to p^* in P. Since the edge weights are non-negative, we have $c_{P^*} \geq 0$. Now, let $\bar{P}$ be the path P, where the subpath P^* is replaced by the single node p^*. Then $c_{\bar{P}} \leq c_P$, so $c_{\bar{P}}$ is also a shortest path. We can analogously replace all cycles by the corresponding nodes. This way, no node is visited more than once and we found the desired path of at most $|V| - 1$ edges. □

2.2.2 Dijkstra's algorithm [8]

If the `Box` in Algorithm 1 is implemented as a priority queue, denoted by Q, with key d_v^*, it is equivalent to Dijkstra's algorithm. The priority queue is an abstract data type which ensures that the element with highest priority, with respect to the key, is on top and can be extracted in constant time. In our case, the node with lowest tentative distance will have the highest priority. Every time a node is pulled from or stored in the priority queue, it has to be reorganised to preserve

[2] Lines 6-11 of Algorithm 1 will be referred to as *scanning* vertex u.

this property. Despite the extra time that is needed to reorganise the queue, this implementation has an important property: after scanning a node v, it can be added to a set that represents the shortest path tree found so far, denoted by $\mathcal{T}$.[3]

Lemma 2.2.3 proves that $\mathcal{T}$ is indeed a shortest path tree. As a result, every node only has to be scanned once. In the case of a P2P problem the algorithm can thus be stopped after the target is added to $\mathcal{T}$. Algorithm 3 states Dijkstra's algorithm for the P2P case. If the priority queue is implemented as Thorup's Fibonacci heap [32], this algorithm has a time complexity of $O(n \log\log n + m)$. Note that this is a better time complexity then that of Algorithm 2.

Algorithm 3 Dijkstra's P2P algorithm

```
1:  procedure DIJKSTRA(s, t)
2:      INIT(s);
3:      Q, T ← {s};
4:      while t ∉ T do
5:          pull u from the top of Q;
6:          for all (u, v) ∈ A do
7:              if (u, v) tense then
8:                  RELAX(u, v);
9:                  Q ← Q ∪ {v};
10:             end if
11:         end for
12:         T ← T ∪ {u};
13:     end while
14: end procedure
```

Lemma 2.2.3. *For every $v \in \mathcal{T}$ during Algorithm 3, $d^*_v = d_{s,v}$ holds.*

Proof. The following proof is by induction on $|\mathcal{T}|$.

1. The tree $\mathcal{T}$ only grows, so $|\mathcal{T}| = 1$ implies $\mathcal{T} = \{s\}$ and $d^*_s = d_{s,s} = 0$ holds.

2. Let u be the last vertex that was scannned and added to $\mathcal{T}$ and define $\mathcal{T}' := \mathcal{T} \setminus \{u\}$. Assume

$$d^*_v = d_{s,v}, \text{ for all } v \in \mathcal{T}'. \tag{I.H.}$$

3. Let P be the shortest path from s to u. If $d^*_u = c_P = d_{s,u}$, then we are done. So, suppose

$$c_P < d^*_u. \tag{2.2.5}$$

   Let $(x, y) \in A$ be the first edge of P such that $x \in \mathcal{T}'$ and $y \notin \mathcal{T}'$ and define P_x the subpath of P from s to x. Then we have

$$c_{P_x} + c_{x,y} \leq c_P \tag{2.2.6}$$

[3]Note that $\mathcal{T}$ is, in fact, not a tree, but a set of nodes. However, a tree can be derived with the edges $(pred(v), v)$, for $v \in \mathcal{T}$. Hence, it will be referred to as a tree.

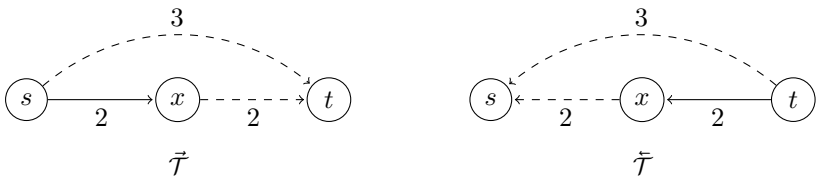


Figure 2.1: Consider the graph $G = (\{s, x, t\}, \{(s, x), (x, t), (s, t)\})$ with $c_{s,x} = 2$, $c_{x,t} = 2$ and $c_{s,t} = 3$. After two forward and two backward iterations of Algorithm 4, we have $\vec{\mathcal{T}} = \{s, x\}$ and $\overleftarrow{\mathcal{T}} = \{x, t\}$. The two trees connect in node x, but $s \to x \to t$ is not the shortest path.

and

$$d_x^* + c_{x,y} \overset{\text{(I.H.)}}{\leq} c_{P_x} + c_{x,y}. \tag{2.2.7}$$

While scanning vertex x, y is relaxed, since they are adjacent. Thus,

$$d_y^* \leq d_x^* + c_{x,y}. \tag{2.2.8}$$

Finally, we have

$$d_u^* \leq d_y^*, \tag{2.2.9}$$

since $y \notin \mathcal{T}'$ and u is the vertex that is pulled after x. By combining inequalities (2.2.5) through (2.2.9) a contradiction follows, which implies that $d_u^* = d_{s,u}$.

Combining steps 1, 2 and 3 proves the lemma. □

2.3 Bidirectional Dijkstra

When solving a P2P problem with a Dijkstra search, $\mathcal{T}$ is grown from the source node s without any sense of 'direction' toward the target node t. The result is a very large search area, i.e. a large number of scanned nodes. The bidirectional Dijkstra search is a method that reduces the search area significantly. Instead of one shortest path tree from s, a second shortest path tree is grown from the target node t.

In order to do this, define $\overleftarrow{A} := \{(v, u) \mid (u, v) \in A\}$ as the set of backward arcs and $\overleftarrow{c}_a := \vec{c}_a$ as the backward weight of edge $a \in \overleftarrow{A}$, for all $a \in \vec{A}$. Every other backward quantity is defined by replacing the edges and weights by the backward edges and backward weights, respectively. Algorithm 4 states the bidirectional Dijkstra algorithm.

It is important to note that, once the two trees connect, say in node $x \in V$, then a shortest s-t path *can* be found. However, Figure 2.1 shows that this path does not necessarily contain x. Using Lemma 2.3.1, it follows that

$$d_{s,t} = \min_{v \in \vec{\mathcal{T}} \cup \overleftarrow{\mathcal{T}}} \left(\vec{d}_v^* + \overleftarrow{d}_v^* \right). \tag{2.3.1}$$

So, after the stopping criterion (Line 3 of Algorithm 4) is met, (2.3.1) will state the shortest path.

Lemma 2.3.1. *If $\vec{\mathcal{T}} \cap \overleftarrow{\mathcal{T}} \neq \emptyset$, then there exists $v \in \vec{\mathcal{T}} \cup \overleftarrow{\mathcal{T}}$ with $\vec{d}_v^* + \overleftarrow{d}_v^* = d_{s,t}$.*

Algorithm 4 Bidirectional Dijkstra

```
1:  procedure BidirDijkstra(s, t)
2:      InitBidir(s, t);
3:      while F⃗ ∩ F⃖ = ∅ do
4:          if Choose(Q⃗, Q⃖) = Q⃗ then
5:              pull u from the top of Q⃗;
6:              F⃗ ← F⃗ ∪ {u};
7:              for all (u, v) ∈ A⃗ do
8:                  if (u, v) tense then
9:                      Relax⃗(u, v);
10:                     Q⃗ ← Q⃗ ∪ {v};
11:                 end if
12:             end for
13:         else
14:             pull u from the top of Q⃖;
15:             F⃖ ← F⃖ ∪ {u};
16:             for all (u, v) ∈ A⃖ do
17:                 if (u, v) tense then
18:                     Relax⃖(u, v);
19:                     Q⃖ ← Q⃖ ∪ {v};
20:                 end if
21:             end for
22:         end if
23:     end while
24: end procedure

function InitBidir(s)
    d⃗*_s, d⃖*_t ← 0;
    pred(s), succ(t) ← ∅;
    Q⃗, F⃗ ← {s};
    Q⃖, F⃖ ← {t};
    for v ∈ V, v ≠ s do
        d⃗*_v ← ∞;
        pred(v) ← ∅;
    end for
    for v ∈ V, v ≠ t do
        d⃖*_v ← ∞;
        succ(v) ← ∅;
    end for
end function
```

Proof. Let $x \in \vec{\mathcal{T}} \cap \overleftarrow{\mathcal{T}}$ be the node that connects the two trees.

First, consider a shortest s-t path P with at least one node $y \notin \vec{\mathcal{T}} \cup \overleftarrow{\mathcal{T}}$. Define subpaths $P_{s\to y}$, $P_{y\to t}$ from s to y and y to t, respectively. Since $x \in \vec{\mathcal{T}}$ and $y \notin \vec{\mathcal{T}}$, we have $d_{s,x} = d^*_x \leq c_{P_{s\to y}}$. Analogously, $d_{x,t} \leq c_{P_{y\to t}}$. So, it follows that

$$d_{s,t} = c_P = c_{P_{s\to y}} + c_{P_{y\to t}} \geq d_{s,x} + d_{x,t} = \vec{d^*_x} + \overleftarrow{d^*_x}. \tag{2.3.2}$$

Since P is a shortest s-t path, we have $\vec{d^*_x} + \overleftarrow{d^*_x} = d_{s,t}$ and the claim holds.

Now, consider any shortest s-t path $P' = (s = v_1, v_2, \ldots, v_\ell = t)$. Let i such that v_i is the last node in P' that also is in $\vec{\mathcal{T}}$. If $v_{i+1} \notin \overleftarrow{\mathcal{T}}$, we have $v_{i+1} \notin \vec{\mathcal{T}} \cup \overleftarrow{\mathcal{T}}$ and thus $\vec{d^*_x} + \overleftarrow{d^*_x} = d_{s,t}$, as proven above. So, assume $v_{i+1} \in \overleftarrow{\mathcal{T}}$. Because $v_i \in \vec{\mathcal{T}}$ and $v_{i+1} \in \overleftarrow{\mathcal{T}}$, we have (Lemma 2.2.3) $\vec{d^*_{v_i}} = d_{s,v_i}$ and $\overleftarrow{d^*_{v_{i+1}}} = d_{v_{i+1},t}$. So

$$d_{s,t} = d_{s,v_i} + c_{v_i,v_{i+1}} + d_{v_{i+1},t} = \vec{d^*_{v_i}} + c_{v_i,v_{i+1}} + \overleftarrow{d^*_{v_{i+1}}}. \tag{2.3.3}$$

When v_{i+1} was added to $\overleftarrow{T}$, v_i was relaxed, ensuring that

$$d_{v_i,t} \leq \overleftarrow{d^*_{v_i}} \leq \overleftarrow{d^*_{v_{i+1}}} + c_{v_i,v_{i+1}} \tag{2.3.4}$$

Substituting (2.3.4) into (2.3.3) results in

$$d_{s,t} = \vec{d^*_{v_i}} + c_{v_i,v_{i+1}} + \overleftarrow{d^*_{v_{i+1}}} \geq \vec{d^*_{v_i}} + \overleftarrow{d^*_{v_i}} \geq d_{s,v_i} + d_{v_i,t} = d_{s,t}. \tag{2.3.5}$$

Hence, the inequalities of (2.3.5) are equalities, resulting in $\vec{d^*_{v_i}} + \overleftarrow{d^*_{v_i}} = d_{s,t}$, which proves the claim.

□

Using Lemma 2.3.1 in Figure 2.1, we find that $\vec{\mathcal{T}} \cup \overleftarrow{\mathcal{T}} = \{s, x, t\}$ and $\vec{d^*_v} + \overleftarrow{d^*_v} = d_{s,t}$, for either $v = s$ or $v = t$.

The Choose function in Line 4 of Algorithm 4 can be specified in different ways.

- Always choose $\vec{Q}$ – This results in a standard Dijkstra search.
- Always choose $\overleftarrow{Q}$ – This results in a standard Dijkstra search from t in $\overleftarrow{G}$.
- Alternate between $\vec{Q}$ and $\overleftarrow{Q}$ – This results in the Dantzig procedure [4].
- Choose the queue with minimal tentative distance – This results in the Nicholson procedure [24].

It is not clear from the literature which of the latter two achieves the best performance.

2.4 The Primal-Dual Algorithm

An alternative way to solve a P2P problem can be found by taking a closer look at Definition 1.0.4. Note that f is an SO, only if it solves (1.0.6) and, more importantly, if f is a flow. Reformulating problem (1.0.6) in terms of decision variables f_a and adding flow constraints results in a linear program that solves the shortest path problem.

The linear program will minimise the total weight

$$z(f) = \sum_{a \in A} c_a(f_a) f_a = \sum_{a \in A} c_a f_a, \tag{2.4.1}$$

while preserving the flow conservation[4] in every node. The flow rate of the O-D pair (s, t) is equal to 1, since we consider a shortest path instance. So, in order for f to be feasible, the following constraints apply.

$$\sum_{u \in V} f_{a_{u,v}} - \sum_{u \in V} f_{a_{v,u}} = \begin{cases} +1 & \text{if } v = s \\ -1 & \text{if } v = t \\ 0 & \text{if } v \in V \setminus \{s, t\} \end{cases} \tag{2.4.2}$$

The LHS of (2.4.2) can be written as the product Mf, where f is the vector $f = (f_{a_1}, \ldots, f_{a_m})$ and $M = (m_{ij})$ is the $(n \times m)$ node-edge incidence matrix of G, defined by

$$m_{ij} = \begin{cases} +1 & \text{if edge } a_j \text{ leaves node } i \\ -1 & \text{if edge } a_j \text{ enters node } i \\ 0 & \text{otherwise} \end{cases}, \quad \text{for} \quad \begin{array}{l} i \in \{1, \ldots, n\}, \\ j \in \{1, \ldots, m\} \end{array}. \tag{2.4.3}$$

So, the primal linear program that solves the shortest path problem is given by

$$\begin{aligned} \text{minimise} \quad & z(f) = \sum_{a \in A} c_a f_a \\ \text{subject to} \quad & m_v f = \begin{cases} +1 & \text{if } v = s \\ -1 & \text{if } v = t \\ 0 & \text{otherwise} \end{cases} \\ & f_a \geq 0, \quad a \in A, \end{aligned} \tag{SP-P}$$

where m_v is the v-th row of M. Note that the $|V|$ equalities in the primal (SP-P) are redundant, since flow conservation in $|V| - 1$ nodes implies the flow conservation in the last node. Hence, any one of the equalities can be left out.

The dual of primal (SP-P) will have the constraints $M_a \pi \leq c_a$, for all $a \in A$, where M_a is defined as the a-th column of the matrix M. Because of the way the matrix M is defined, this reduces to $\pi_u - \pi_v \leq c_{u,v}$ for all $(u, v) \in A$. By leaving out the equality of row t of primal (SP-P), we may take $\pi_t = 0$. So the dual of the shortest path problem is then given by

$$\begin{aligned} \text{maximise} \quad & \pi_s \\ \text{subject to} \quad & \pi_u - \pi_v \leq c_{u,v}, \quad (u, v) \in A. \\ & \pi_v \lesseqgtr 0, \quad v \in V \end{aligned} \tag{SP-D}$$

Because the shortest path problem can be formulated as a linear program, there is a broad spectrum of algorithms that can solve it. Below we will discuss the Primal-Dual Algorithm. The Primal-Dual Algorithm is a general LP solving algorithm that works particularly well on network related problems. First we will discuss the classic Primal-Dual Algorithm, after which we will apply it to the primal-dual pair (SP-P) and (SP-D).

[4]Note that, for a P2P instance, the flow conservation of edge flows does hold, since there is only one O-D pair.

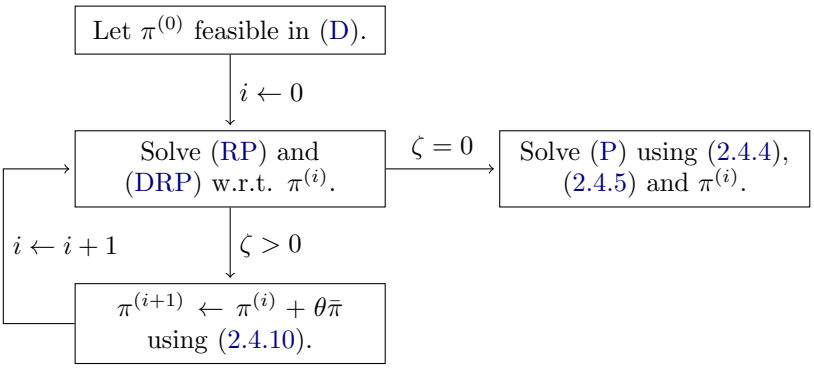


Figure 2.2: Schematic representation of the Primal-Dual Algorithm.

2.4.1 The classic Primal-Dual Algorithm

As the name suggests, the Primal-Dual Algorithm uses both the primal and the dual of a linear program to solve it. Consider the following primal and dual.

$$\begin{array}{lll} \min & z = c^T x & \\ \text{s.t.} & Mx = b \geq 0, & \text{(P)} \\ & x \geq 0. & \end{array} \qquad \begin{array}{lll} \max & w = b^T \pi & \\ \text{s.t.} & M^T \pi \leq c, & \text{(D)} \\ & \pi \lneqq 0. & \end{array}$$

We assume that (P) is feasible and $c \geq 0$. The former assumption is plausible, since (P) will represent program (SP-P), which is feasible by Assumption 1.0.1. The latter will provide us with an initial feasible dual solution $\pi = 0$ and is plausible, since c will represent the non-negative edge weights of a graph.

A well-known property of linear programs is the *complementary slackness condition*, which states: x feasible in (P) and π feasible in (D) are both optimal if and only if the following equations hold.

$$\pi_i(m_i^T x - b_i) = 0 \quad \text{for all } i \in \{1, \ldots, n\}, \tag{2.4.4}$$
$$(c_j - M_j^T \pi)x_j = 0 \quad \text{for all } j \in \{1, \ldots, m\}, \tag{2.4.5}$$

with M_j the j-th column and m_i the i-th row of M.

The Primal-Dual Algorithm starts with a feasible π for (D). It will then attempt to find a feasible x for (P) that obeys the complementary slackness conditions. If it succeeds, an optimal solution is found. Otherwise, it is able to construct a new feasible dual solution that improves the objective function value. Eventually, by iterating this process, a feasible primal-dual solution pair will be found that obeys the complementary slackness conditions. A schematic representation of this process is given in Figure 2.2.

Let π be feasible in (D). Note that the conditions of (P) are equalities, so (2.4.4) is satisfied for any feasible x. Hence, we focus on (2.4.5). If a feasible x can be found, with $x_j = 0$, whenever $c_j - M_j^T \pi > 0$, then (2.4.5) is satisfied and a solution is found. The Primal-Dual Algorithm will attempt to find such x, given a feasible π, by solving an auxiliary problem, called the restricted primal (RP).

Define

$$J := \{j : M_j^T \pi = c_j\} \tag{2.4.6}$$

as an index set and consider

$$\begin{aligned} \min \quad & \zeta = \sum_{i=1}^{n} y_i \\ \text{s.t.} \quad & \sum_{j \in J} m_{ij} x_j + y_i = b_i, \quad i \in \{1, \ldots, n\}, \\ & x_j \geq 0, \quad j \in J, \\ & x_j = 0, \quad j \notin J, \\ & y_i \geq 0, \end{aligned} \tag{RP}$$

where y_i is a new variable for each equality constraint in (P).

The program (RP) is then solved. If a solution $(\bar{x}, \bar{y})$ is found with $\zeta = 0$, then $\bar{x}$ and π will obey the complementary slackness conditions and we are done. If not, we update the current feasible solution π of (D) to one that increases the corresponding objective function value w. In order to do this, consider the dual of the restricted primal.

$$\begin{aligned} \max \quad & \omega = b^T \pi \\ \text{s.t.} \quad & M_j^T \pi \leq 0, \quad j \in J, \\ & \pi_i \leq 1, \quad i \in \{1, \ldots, n\}, \\ & \pi_i \lesseqgtr 0, \end{aligned} \tag{DRP}$$

Let $\bar{\pi}$ denote the optimal solution of (DRP). The original π is then replaced by

$$\pi \leftarrow \pi + \theta \bar{\pi}, \tag{2.4.7}$$

where θ will be chosen in such a way that the new π increases w while remaining feasible in (D).

Proposition 2.4.1. *Let $\bar{\zeta} > 0$ be the optimal value of (RP) and let $\bar{\pi}$ denote the optimal solution to (DRP). Then there exists $\theta > 0$, such that $\pi^* := \pi + \theta \bar{\pi}$ is feasible in (D) with $b^T \pi^* > b^T \pi$.*

Proof. Since $\bar{\pi}$ is the optimal solution to (DRP), we have $b^T \bar{\pi} = \bar{\zeta} > 0$. So, the objective function of (D) for our new π^* will then satisfy

$$b^T \pi^* = b^T \pi + \theta b^T \bar{\pi} = b^T \pi + \theta \bar{\zeta} > b^T \pi, \tag{2.4.8}$$

for $\theta > 0$. To remain feasible, we require, for all $j \in \{1, \ldots, m\}$,

$$M_j^T \pi^* = M_j^T \pi + \theta M_j^T \bar{\pi} \leq c_j. \tag{2.4.9}$$

It suffices to show the following.

- For $j \in J$, we have $M_j^T \pi \leq c_j$, since π is feasible in (D), and $M_j^T \bar{\pi} \leq 0$, since $\bar{\pi}$ is feasible in (DRP). Hence, (2.4.9) holds for all $j \in J$.
- For $j \notin J$, we have $M_j^T \pi < c_j$ by the definition of J. So, by choosing[5]

$$\theta := \min_{\{j \notin J \mid M_j^T \bar{\pi} > 0\}} \left(\frac{c_j - M_j^T \pi}{M_j^T \bar{\pi}} \right), \tag{2.4.10}$$

[5]Note that $\theta > 0$. Also note that, if $M_j^T \bar{\pi} \leq 0$ for all $j \in \{1, \ldots, m\}$, then θ could be increased indefinitely, resulting in an infeasible (P), which contradicts our assumption. Hence, there exists $j \notin J$, such that $M_j^T \bar{\pi} > 0$.

requirement (2.4.9) will hold for all $j \notin J$.

The $\theta > 0$ given in (2.4.10) satisfies the requirements of Proposition 2.4.1. □

In order to see that the Primal-Dual Algorithm terminates in a finite number of steps, consider the extended version of (RP), given by

$$\begin{aligned} \min \quad & \zeta^* = \sum_{i=1}^{n} y_i \\ \text{s.t.} \quad & Mx + y = b, \\ & x, y \geq 0. \end{aligned} \tag{2.4.11}$$

Note that a basis of (RP) is also a basis of (2.4.11). Program (2.4.11) has a finite number of basic feasible solutions, since $n, m < \infty$. Every iteration a $j^* \notin J$ is chosen according to (2.4.10). By (2.4.9), j^* will enter J at the next iteration. As a result, x_{j^*} enters the basis of (2.4.11), resulting in a new feasible solution with a strictly lower value, since $-M_{j^*}^T \bar{\pi} < 0$. Hence, no cycling will take place, ensuring a finite number of iterations before termination.

Note that, in every iteration, the Primal-Dual algorithm requires an optimal solution for programs (RP) and (DRP). In fact, the Primal-Dual algorithm will only be effective if either of the programs allow an easy solution. The structure of (DRP) will provide an easy solution if the Primal-Dual Algorithm is applied to the shortest path problem, as will be shown next.

2.4.2 Application to the shortest path problem

For convenience, the Primal-Dual Algorithm will be applied to a backward P2P instance. The primal for such a backward shortest path instance is

$$\begin{aligned} \text{minimise} \quad & z(f) = \sum_{a \in \overleftarrow{A}} \overleftarrow{c}_a(f_a) f_a = \sum_{a \in \overleftarrow{A}} \overleftarrow{c}_a f_a \\ \text{subject to} \quad & \overleftarrow{m}_v f = \begin{cases} -1 & \text{if } v = s, \\ +1 & \text{if } v = t, \\ 0 & \text{otherwise,} \end{cases} \\ & f_a \geq 0, \quad a \in \overleftarrow{A}. \end{aligned} \tag{BSP-P}$$

By omitting the equation of row s, such that $\pi_s = 0$, we get the dual of the backward shortest path instance.

$$\begin{aligned} \text{maximise} \quad & \pi_t \\ \text{subject to} \quad & \pi_u - \pi_v \leq \overleftarrow{c}_{u,v}, \quad (u,v) \in \overleftarrow{A}, \\ & \pi_v \lesseqgtr 0, \quad v \in V. \end{aligned} \tag{BSP-D}$$

Let $\pi^{(0)}$ be an initial feasible solution for the dual (BSP-D). Every iteration we construct the index set

$$\overleftarrow{J} \quad = \quad \{(u,v) \in \overleftarrow{A} \mid \pi_u^{(i)} - \pi_v^{(i)} = \overleftarrow{c}_{u,v}\}. \tag{2.4.12}$$

The restricted primal is then given by

$$\begin{aligned} \text{minimise} \quad & \zeta = \sum_{i=1}^{m-1} y_i \\ \text{subject to} \quad & \overleftarrow{m}_v f + y_v = \begin{cases} +1 & \text{if } v = t, \\ 0 & \text{otherwise,} \end{cases} \\ & f_{u,v} \geq 0, \quad (u,v) \in \overleftarrow{J}, \\ & f_{u,v} = 0, \quad (u,v) \notin \overleftarrow{J}, \\ & y_i \geq 0. \end{aligned} \tag{BSP-RP}$$

Finally, the dual of (BSP-RP) is

$$\begin{aligned} \text{maximise} \quad & \pi_t \\ \text{subject to} \quad & \pi_u - \pi_v \leq 0, \quad (u,v) \in \overleftarrow{J}, \\ & \pi_u \leq 1, \\ & \pi_u \lneqq 0. \end{aligned} \tag{BSP-DRP}$$

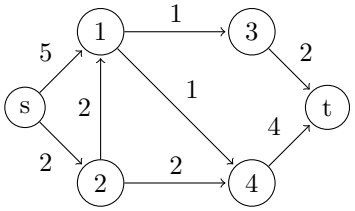
Finding an optimal solution for (BSP-DRP) is easy. We have $\pi_s = 0$, so set $\bar{\pi}_u \leftarrow 0$ for every u that has a path from u to s in $\overleftarrow{J}$ to satisfy $\pi_u - \pi_v \leq 0$, for $(u,v) \in \overleftarrow{J}$. If there is a path from t to s in $\overleftarrow{J}$, then $\bar{\pi}_t = 0$ is optimal and $\zeta = 0$, so we are done. If not, we maximise by setting $\bar{\pi}_u \leftarrow 1$, for all nodes $u \in V$ that are not already set to 0, while satisfying $\pi_u \leq 1$ and $\pi_u - \pi_v \leq 0$, for $(u,v) \in \overleftarrow{J}$. Note that it is not necessary, for maximising (BSP-DRP), to set $\bar{\pi}_u \leftarrow 1$, for $u \neq t$. This will, however, illustrate the the progress of this algorithm more clearly, as can be seen in Example 2.4.2.

Now, $\bar{\pi}$ is an optimal solution to (BSP-DRP) and $\pi^{(i)}$ can be updated by

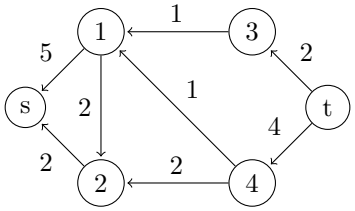
$$\pi^{(i+1)} \leftarrow \pi^{(i)} + \theta\bar{\pi}, \text{ where } \theta = \min_{\{(u,v) \notin \overleftarrow{J} \mid \bar{\pi}_u - \bar{\pi}_v > 0\}} [\overleftarrow{c}_{u,v} - (\pi_u - \pi_v)]. \tag{2.4.13}$$

Further thought reveals that, once $\bar{\pi}_u = 0$, it will remain 0 and the value of π_u will therefore remain fixed until the algorithm terminates. So, in every iteration, only the edges from nodes that are next closest to s will be added to $\overleftarrow{J}$. In fact, if $\pi^{(0)} \equiv 0$, it can be shown that this algorithm is equivalent to Dijkstra's Algorithm [34]. Example 2.4.2 shows the primal dual algorithm applied to a P2P instance.

Example 2.4.2. Consider the following instance $\vec{G}_{(s,t,1),\vec{c}}$, where the cost $\vec{c}$ is stated besides the arcs, and the corresponding instance $\overleftarrow{G}_{(s,t,1),\overleftarrow{c}}$.



Instance $\vec{G}_{(s,t,1),\vec{c}}$.



Instance $\overleftarrow{G}_{(s,t,1),\overleftarrow{c}}$.

Because the equality of row s is omitted, we have $\pi_s = 0$ and we define $\bar{\pi} := (\bar{\pi}_1, \bar{\pi}_2, \bar{\pi}_3, \bar{\pi}_4, \bar{\pi}_t)$. Let $\pi^{(0)} = (0, 0, 0, 0, 0)$ be the initial feasible solution to the dual. The Primal-Dual Algorithm will proceed with the following iterations.

1. $\overleftarrow{J} \leftarrow \emptyset$,
   $\bar{\pi} \leftarrow (1, 1, 1, 1, 1)$,
   $\theta \leftarrow 2$, for arc $(2, s)$,
   $\pi^{(1)} \leftarrow (2, 2, 2, 2, 2)$.

2. $\overleftarrow{J} \leftarrow \{(2, s)\}$,
   $\bar{\pi} \leftarrow (1, 0, 1, 1, 1)$,
   $\theta \leftarrow 2$, for arcs $(4, 2)$ and $(1, 2)$,
   $\pi^{(2)} \leftarrow (4, 2, 4, 4, 4)$.

3. $\overleftarrow{J} \leftarrow \{(2, s), (1, 2), (4, 2)\}$,
   $\bar{\pi} \leftarrow (0, 0, 1, 0, 1)$,
   $\theta \leftarrow 2$, for arc $(3, 1)$,
   $\pi^{(3)} \leftarrow (4, 2, 5, 4, 5)$.

4. $\overleftarrow{J} \leftarrow \{(2, s), (1, 2), (4, 2), (3, 1)\}$,
   $\bar{\pi} \leftarrow (0, 0, 0, 0, 1)$,
   $\theta \leftarrow 2$, for arc $(t, 3)$,
   $\pi^{(4)} \leftarrow (4, 2, 5, 4, 7)$.

5. $\overleftarrow{J} \leftarrow \{(2, s), (1, 2), (4, 2), (3, 1), (t, 3)\}$,
   $\bar{\pi} \leftarrow (0, 0, 0, 0, 0)$,
   $\zeta \leftarrow 0$.

The optimal solution is $\pi^{(4)} = (4, 2, 5, 4, 7)$ with the corresponding set $\overleftarrow{J} = \{(2, s), (1, 2), (4, 2), (3, 1), (t, 3)\}$. The shortest path from s to t can be found in the forward set $\vec{J} = \{(u, v) \mid (v, u) \in \overleftarrow{J}\}$, i.e. $s \to 2 \to 1 \to 3 \to t$, with $d_{s,t} = \pi_t^{(4)} = 7$. Note that the tree, implied by the set $\vec{J}$, grows equivalently to the tree $\mathcal{T}$ of a Dijkstra run.

3. Speed-up techniques

This chapter will contain several speed-up techniques for P2P instances, that exploit the staticity of graphs. In many applications that use shortest path computations, a large number of queries is being made on the same graph. For example, a road network is a reasonably static graph. Many queries for optimal routes can be made before the graph has to be updated. This staticity can be exploited by *preprocessing* the graph, *storing* the preprocessed information and using it to reduce *query*-time. Such a query consists of one s-t pair, for which the shortest path has to be found. The information from the preprocessing phase is used to accelerate the search. Once a graph changes, it has to be preprocessed again.

Ideally both preprocessing-time and query-time are short, while storing little data. In practice, there usually is a trade-off between them, which results in application-dependent performance. More extensive preprocessing will, for example, often result in faster query-times. However, fast changing networks, which have to be preprocessed more often, could benefit more from a less extensive preprocessing phase, with longer query-times as a result. Using a large amount of data can result in both fast preprocessing-time and query-time. As a consequence, queries can only be made from devices that have fast access to a large amount of data. The performance of the speed-up techniques in this chapter will not be addressed, since this is application-dependent.

The various techniques discussed in this chapter will all try to find information, during preprocessing, that can be used to prioritise vertices that are more likely to be on the shortest path and exclude vertices that are, with certainty, not in the shortest path. Every technique will have an alternative way of obtaining and using this information.

3.1 Goal directed techniques

The classical Dijkstra algorithm is, as mentioned in section 2.3, a very robust algorithm which searches in all directions until it finds the target. In the preprocessing phase of the following goal directed speed-up techniques, information is computed that will guide the query-search toward the target. This way, the search space is reduced, which will lead to faster query-times.

3.1.1 Geometric Containers [33]

In the preprocessing phase of the Geometric Containers method, a set $V_{u,v} \subset V$ is computed, for every edge $(u, v) \in A$. The set $V_{u,v}$ contains the vertices w for which a shortest u-w path exists, that has (u, v) as the first edge of that path. So

$$V_{u,v} := \{w \in V \mid \exists \text{ shortest path } (u, v, \ldots, w)\}. \tag{3.1.1}$$

Storing a set of vertices for every edge in A uses a lot of storage space. Instead, the sets are encoded, using geometric information. For every set $V_{u,v}$, a geometric container $S_{u,v}$ is stored, such that every node in $V_{u,v}$ lies in the container $S_{u,v}$. In order to do this, we assume that we are given a layout $L : V \to \mathbb{R}$ in the Euclidean plane. There are numerous ways to encode these geometric

containers. In many cases the bounding box is the most efficient way to encode $S_{u,v}$, according to [33]. For the bounding box, only four corner points have to be stored, instead of $V_{u,v}$. Note that $S_{u,v}$ may contain more nodes than $V_{u,v}$ alone. However, this will not affect the correctness of the query algorithm.

The query algorithm is a slightly altered Dijkstra algorithm. Normally, while scanning a node $u \in V$, the node v is added to Q if the edge (u, v) is tense. In this case, v is not added to Q if $t \notin S_{u,v}$. In other words, the Dijkstra search is pruned at edge $(u, v) \in A$ if there does not exist a shortest u-t path that starts with (u, v). This algorithm can be easily combined with the bidirectional A* algorithm that will be discussed next.

3.1.2 A* search and ALT

For a shortest s-t path query, the A* algorithm defines a heuristic $h : V \to \mathbb{R}$ by the following properties:

- Feasibility:
$$c_{v,w} - h_v + h_w \geq 0, \quad \forall (v, w) \in A. \tag{3.1.2}$$
- Let $h_t = 0$. It is then easy to prove that $h_u \leq d_{u,t}$, for all $u \in V$. Hence, h_u is a lower bound on $d_{u,t}$.

Given such a heuristic, new edge weights can be defined as

$$c^h_{v,w} := c_{v,w} - h_v + h_w, \quad \forall (u, v) \in A. \tag{3.1.3}$$

The feasibility constraint on the heuristic ensures non-negativity of the new edge weights. Furthermore, for every possible s-t path $P \in \mathcal{P}$ we have

$$\begin{aligned} c^h_P &= \sum_{(v,w)\in P} c^h_{v,w} && (3.1.4)\\ &= \sum_{(v,w)\in P} c_{v,w} - h_v + h_w && (3.1.5)\\ &= \left(\sum_{(v,w)\in P} c_{v,w} \right) - h_s + h_t && (3.1.6)\\ &= c_P - h_s. && (3.1.7) \end{aligned}$$

So, the length of every s-t path is only reduced by the constant h_s. As a result, the shortest path in the graph with adjusted edge weight is also the shortest path in the original graph. Finding the shortest path comes down to a Dijkstra search in the graph with the new edge weight c^h.[1] The new edge weights[2] have as a result that edges, that lead to nodes which are closer to the target, weigh less. Therefore, the nodes that are closer to the target will be higher in the priority queue and will thus be scanned earlier. The shortest path tree $\mathcal{T}$ will grow more toward the target, instead of growing in arbitrary direction.

[1] This is equivalent to a Dijkstra search with priority queue key $d^*_u + h_u$ and the original edge weight c, as is shown in [13].

[2] Or altered priority queue key.

Surprisingly, the Primal-Dual Algorithm (see Section 2.4) is also equivalent, if the initial feasible dual is chosen as $\pi^{(0)} := -h$. Recall that such an initial solution is only feasible if the condition

$$\pi_u^{(0)} - \pi_v^{(0)} \leq \overleftarrow{c}_{u,v}, \quad \forall (u,v) \in \overleftarrow{A} \tag{3.1.8}$$

holds. Substituting the heuristic and the using the fact that $\overleftarrow{c}_{u,v}$, for $(u,v) \in \overleftarrow{A}$, is equal to $\vec{c}_{v,u}$, for $(v,u) \in \vec{A}$, results in

$$h_v - h_u \leq c_{v,u}, \quad \forall (v,u) \in \vec{A}. \tag{3.1.9}$$

Interchanging (v,u) for (v,w) results in inequality (3.1.2). So, for every feasible heuristic, $\pi^{(0)} := -h$ is indeed an initial feasible solution.

The equivalence between the Primal-Dual Algorithm and the A* algorithm is proven by Xugang Ye et al. [34].

Choosing a heuristic with ALT

Heuristics with values h_u that are sharper lower bounds on $d_{u,t}$, for all $u \in V$, will have a faster Dijkstra search. In theory, if the heuristic is an exact lower bound (i.e. $h_u = d_{u,t}$ for all $u \in V$), only nodes on the shortest s-t path will be visited during the Dijkstra search.

A first choice for a heuristic h_u could be the metric distances between u and t, after the graph is projected on a metric space, such that the distance between adjacent nodes is equal to the weight of the edge. This heuristic is then feasible by the triangle inequality. Although this results in an algorithm that is a lot faster than classic Dijkstra, it performs poorly compared to the other speed-up techniques. ALT (A*, Landmarks and the Triangle inequality) is a speed-up technique that defines a heuristic that has tighter lower bounds on $d_{u,t}$, with better performance as a result.

In the preprocessing phase, define $L \subset V$ as a set of landmarks and, for every $l \in L$, compute $d_{u,l}$ and $d_{l,u}$, for all $u \in V$. These distances will be used during the query phase to define the heuristic. For an s-t query, a subset $L^* \subset L$ of landmarks is chosen for which the following lower bounds are made, for every vertex $v \in V$, using the triangle inequality.

- $\forall l \in L^*$ we have $d_{v,l} - d_{t,l} \leq d_{v,t}$.
- $\forall l \in L^*$ we have $d_{l,t} - d_{l,v} \leq d_{v,t}$.

Now let $h_v := \max_{l \in L^*} \{d_{l,t} - d_{l,v}, d_{v,l} - d_{t,l}\}$. This heuristic h is feasible, since the maximum of two feasible heuristics is also feasible. Choosing the right subset of landmarks is very important for good performance. A big subset requires a lot of calculation to determine the heuristic, whereas a small subset results bounds that are less tight. Best results are achieved by choosing small subsets of landmarks that lay behind the source and target node at the edge of the graph.

Bidirectional A*

It is possible to do a bidirectional A* search. For this a forward heuristic $\vec{h}$ and backward heuristic $\overleftarrow{h}$ are defined during the query phase. Unfortunately,

a straightforward bidirectional search, using these heuristics, is in general not sufficient. Suppose the two searches meet in the node $u \in V$. The combined length of the forward and backward search is then equal to

$$\vec{d}^{h}_{s,u} + \overleftarrow{d}^{h}_{u,t} = d_{s,u} + d_{u,t} - (\vec{h}_s + \overleftarrow{h}_s) + (\vec{h}_u + \overleftarrow{h}_u). \tag{3.1.10}$$

This distance is dependent on the node u, so it is not guaranteed that this results in the shortest path.

We call $\vec{h}$ and $\overleftarrow{h}$ consistent if $\vec{h}_u + \overleftarrow{h}_u$ is constant for all $u \in V$. In this case the total found distance will be constant, so differences in distance will be equivalent to differences in distance in the original graph. In order to guarantee shortest paths, one can use consistent heuristics. This is, however, not always an ideal choice. For example, ALT is in general not a consistent heuristic.

Another way to guarantee shortest paths is altering the stopping criterion in the following way. In the initialisation of the query algorithm, define length of the current best found s-t distance as $\mu := \infty$ and the meeting point of the current best found s-t path as u_μ. If the forward and backward search meet in a vertex u let $\vec{P}_{s,u}$ and $\overleftarrow{P}_{u,t}$ be the paths of the forward and backward search, respectively. If $\vec{c}_{\vec{P}_{s,u}} + \overleftarrow{c}_{\overleftarrow{P}_{u,t}} < \mu$ holds, do the following:

- Update $\mu \leftarrow \vec{c}_{\vec{P}_{s,u}} + \overleftarrow{c}_{\overleftarrow{P}_{u,t}}$.
- Update $u_\mu \leftarrow u$.
- Prune both the forward and backward search at u.

The algorithm can now be stopped if either u is scanned with a distance larger than μ or if $\vec{Q} \cup \overleftarrow{Q} = \emptyset$. The shortest s-t path then has length μ with meeting point u_μ. This stopping criterion will be referred to as the *extended stopping criterion*.

3.1.3 Arc Flags [16]

During the preprocessing phase of Arc Flags, the graph is partitioned into k (preferably balanced) disjoint cells, $\mathcal{K} := \{K_1, \ldots, K_k\}$, with a small number of boundary nodes[3]. Every edge $a \in A$ is assigned a label of k bits. The i-th bit of this label will be set if and only if a is the beginning of any shortest path to at least one node in K_i.

The label of arc $(u, v) \in A$ can be computed by performing a Dijkstra search, for both u and v, to all other nodes in V[4]. For each $w \in V$, if $|d_{u,w} - d_{v,w}| = c_{u,v}$, then (u, v) is the start of a shortest path to w. The bit of the (u, v)-label corresponding to the cell of w can now be set. This method takes $2m$ one-to-all Dijkstra searches, which can be improved.

For every boundary node of cell K_i, perform a one-to-all backward Dijkstra search. At the end of each search, all the i-th bits of the edges in the corresponding shortest path tree can be set. This way, only one one-to-all search has to be performed for every boundary node. Moreover, the searches of all boundary nodes of a cell can be performed simultaneously by updating the tentative distance of all boundary nodes at once, taking into account the mutual

[3] A boundary node is a node that has at least one adjacent node that is not in the same cell.

[4] Such searches are referenced to as one-to-all searches.

distances of the boundary nodes. Note that contracting all nodes of a cell into one super node and performing one backward search from this super node can result in incorrect arc-flags.

A search for an s-t query can now simply prune the search at an arc a, if the bit of its label, corresponding to the cell of target node t, is not set. In this case there does not exist any shortest path starting from a to the cell of node t, hence a search in that direction is unnecessary.

To improve performance, a multilevel partitioning can be made. Here every cell will again be partitioned into several smaller cells and lower level arc flags will be calculated for every edge in such a partitioned cell. The query search can then be pruned according to the lower level arc flags, once it reaches a cell of the target node to reduce the search area within this cell. This multilevel arc flag technique works particularly well in combination with a bidirectional search. This way, the inefficient lower level searches can be kept to a minimum, since the forward and backward search could meet before either search has reached the cell of the opposite search.

3.1.4 Precomputed Cluster Distances (PCD) [22]

Again, during the preprocessing phase, the graph is partitioned into k (balanced) cells, $\mathcal{K} := \{K_1, \ldots, K_k\}$, with preferably a small number of boundary nodes. Next, as the name suggests, the distances between these cells will be computed, where the distance from cell K_i to K_j is defined as

$$d_{K_i,K_j} := \min_{u \in K_i, v \in K_j} d_{u,v}, \quad \forall i,j \in \{1,...,k\}. \tag{3.1.11}$$

Note that in general $d_{K_i,K_j} \neq d_{K_j,K_i}$, since we have a directed graph. In order to compute the distance from cell K_i to K_j, add a new node s_i^* to the graph and add new edges (s_i^*, u) of weight $c_{s_i^*,u} := 0$, for all boundary nodes u of cell K_i. In other words, let

$$V_i^* := V \cup \{s_i^*\}, \tag{3.1.12}$$

$$A_i^* := A \cup \{(s_i^*, u) \mid u \text{ is a boundary node of } K_i\} \tag{3.1.13}$$

and, for all $a \in A^*$,

$$c_a^* = \begin{cases} c_a & \text{if } a \in A, \\ 0 & \text{otherwise.} \end{cases} \tag{3.1.14}$$

Using the new graph $G_i^* = (V_i^*, A_i^*)$, with weight function c^*, start a Dijkstra search from node s_i^* and terminate the search once a node from K_j is scanned, say w. We then have $d_{K_i,K_j} = d_{s_i^*,w}$.

The s-t query consists of a bidirectional Dijkstra search. The precomputed cluster distances will be used to prune the search on certain nodes. The conditions for such a prune will be established below for the forward search. Analogous conditions can be made for the backward search.

After scanning a node u in the forward search, a lower bound on the minimal distance from s, via u, to t, denoted $d_{s,u,t}$, is computed. Note that

$$d_{s,u,t} = d_{s,u} + d_{u,t} \geq d_{s,t}. \tag{3.1.15}$$

This lower bound is then compared to an upper bound on $d_{s,t}$. The search is be pruned whenever the lower bound of the scanned node exceeds the current upper bound. The computation of these bounds is discussed below.

For an s-t query, two values have to be determined before the bounds can be computed. Let S and T be the cells of s and t respectively. From both nodes an individual Dijkstra search is performed until the closest boundary nodes $s' \in S$ and $t' \in T$ of the corresponding cluster are found. After $d_{s,s'}$ and $d_{t',t}$ are found, the bounds will be computed and the search will be pruned accordingly.

Computing the lower bounds

For every scanned node u a new lower bound is computed. Define U and T as the cell of node u and t, respectively. Next, define $t' \in T$ as the boundary node that is closest to t. It is then easy to verify that the following inequality holds

$$d_{s,u} + d_{U,T} + d_{t',t} < d_{s,u,t}. \tag{3.1.16}$$

The quantities $d_{s,u}$, $d_{U,T}$ and $d_{t',t}$ are known, since node u is already scanned by the forward search, $d_{U,T}$ is a precomputed cluster distance and $d_{t',t}$ is already found. The following lower bound can thus be defined

$$\underline{d}_{s,u,t} := d_{s,u} + d_{U,T} + d_{t',t}. \tag{3.1.17}$$

Computing the upper bounds

Let $k_i^* \in K_i$ be the starting point and $t^* \in T$ be the end point of the shortest path from cell K_i to cell T. Then the following inequality holds for every $k_i \in K_i$:

$$d_{s,t} \leq d_{s,k_i^*} + d_{K_i,T} + d_{t^*,t}. \tag{3.1.18}$$

Whenever the forward search scans such a starting node k_i^* for cluster K_i, the current upper bound can be updated. The values d_{s,k_i^*} and $d_{K_i,T}$ are, again, already known. If the backward search has already visited t^*, $d_{t^*,t}$ is also known. However, if that is not the case, $d_{t^*,t}$ can be upper bounded by the diameter of T, i.e.

$$d_{t^*,t} \leq \max_{u,v \in T} d_{u,v}. \tag{3.1.19}$$

The current upper bound $\bar{d}_{s,t}$ can thus be updated to either $d_{s,k_i^*} + d_{K_i,T} + d_{t^*,t}$ or $d_{s,k_i^*} + d_{K_i,T} + \max_{u,v \in T} d_{u,v}$.

Now, whenever $\underline{d}_{s,u,t} > \bar{d}_{s,t}$, the search can be pruned. Note that the forward search can use both the lower and upper bounds of the backward search and vice versa.

3.2 Hierarchy-Based Methods

The following techniques exploit the natural hierarchy that a lot of networks have. They will aim to prioritize on "important" nodes and arcs, that are on many shortest paths. For example, in road networks, highways can be seen as important arcs of the network, since these arcs are on many shortest (or fastest) paths.

3.2.1 Contraction

Intuitively it is efficient to add shortcuts between important vertices amongst which many long-distance shortest paths lay. The following speed-up techniques use this idea.

Contraction Hierarchies (CH) [12]

The vertices are numbered $1, \ldots, n$ in ascending order of 'importance'. This notion of importance will be discussed below. The vertices will be repeatedly contracted in order of importance and shortcuts will be added to preserve shortest paths.

Let v be the least important node that will be contracted next. For all possible node pairs $u, w \in \{v+1, \ldots, n\}$, where u and w have higher importance than v and are neighbours of v, a local search is performed. If the shortest path from u to w is unique and contains v, a shortcut edge (u, w) with weight $d_{u,w}$ is added to the graph. After all such pairs are checked and shortcuts are added accordingly, v and all its in- and outgoing edges are temporarily removed from the graph. The contraction continues on the remaining graph. Figure 3.1 shows the contraction of such a node v.

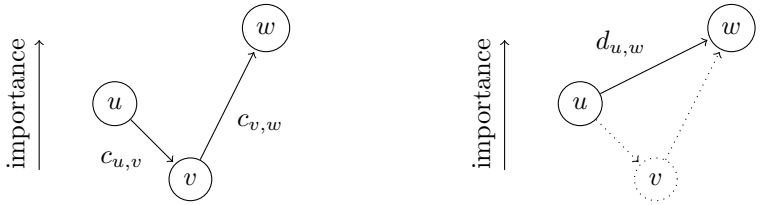


Figure 3.1: Contraction of the node v.

For an s-t query, all contracted vertices and edges are replaced. The graph now contains a 'hull' of upward shortcuts that lead to more important nodes. A bidirectional Dijkstra search from s and t is performed in this extended graph. The search will be limited to only visit vertices of higher importance. It is easy to see that, due to the way the shortcuts are added, such a search indeed finds the shortest path. The search is terminated whenever the extended stopping criterion is met, see page 24.

How important a vertex is, can be defined in numerous ways. In [12] a priority queue is used that gives priority to vertices according to a linear combination of several terms. This priority queue will be maintained and updated during the contraction. Arguably the most important term is the edge difference, defined as: The number of shortcuts introduced when contracting v minus the number of edges incident to v.

Highway Hierarchies (HH) [28, 30]

With this method multiple layers of 'highway' arcs are added and unimportant/'local' arcs and vertices are contracted during the preprocessing phase to create a highway hierarchy. This can be achieved in the following way. Make a Dijkstra-ranking, where $N_{u,v}$ represents the total number of scanned nodes, after v is scanned, in a Dijkstra search from u. For a parameter H, find

$v \in V$ such that $N_{u,v} = H$ and let $d_u^H := d_{u,v}$. Then define a neighbourhood $\mathcal{N}_u^H := \{w \in V \mid d_{u,w} \leq d_u^H\}$. A highway hierarchy is recursively constructed as follows, where $G_0 := G_0^* := (V_0, A_0)$ is the original graph.

- Let A_i be the subset of all arcs $(u, v) \in A_{i-1}^*$, for which there exists a shortest path $(x, \ldots, u, v, \ldots, y)$ between any pair $x, y \in V_{i-1}^*$, with $v \notin \mathcal{N}_x^H$ and $u \notin \mathcal{N}_y^H$. Let V_i be the maximum subset of V_{i-1}, such that, together with edges A_i, there are no isolated nodes. Then $G_i := (V_i, A_i)$ is called the i-th layer.
- The layer G_i is modified, to a new graph G_i^*, in two steps.
  - First the *2-core* of G_i is taken. This is done by repeatedly removing all nodes of degree one, together with its adjacent edge. The 2-core is the graph that remains. This way, all trees that where connected to the graph are removed and every node in the 2-core has a degree of two or more.
  - All *lines* in the remaining 2-core are replaced by edges. A line is a path $(u_0, \ldots, u_\ell)$, where the inner nodes $(u_1, \ldots, u_{\ell-1})$ are of degree two. A line is replaced by removing all edges and inner nodes of the path and replacing it by an edge with weight equal to the length of the original path.

  All the lines and trees that are removed are referred to as the *components* of G_i. The remaining graph, that only has nodes of degree three or higher, is denoted $G_i^* = (V_i^*, A_i^*)$ and is referred to as the *core* of G_i. The next graph layer will be based on G_i^*.

After all layers have been constructed, they are joined together into one multi-layered graph. Note that the graphs G_i^* are only constructed to define the next layer, but will not be part of the multi-layered graph. For all nodes $v_i \in V_i$, edges (v_{i-1}, v_i) of weight zero are created, where v_{i-1} is the node in V_{i-1} that v_i is a copy of.

Again, as query algorithm, a slightly altered bidirectional upward Dijkstra search is performed. The search is altered in the following two ways.

- We define the *entrance point* of layer i as the first node of its core that has been scanned by a search. A search will only scan the nodes that belong to the layer i neighbourhood of the entrance point of that layer.
- Nodes of components are only scanned if the edge from its predecessor to that node is an upward edge or if that edge also belongs to the component.

The search is only terminated if $\vec{Q} \cup \overleftarrow{Q} = \emptyset$. The restrictions on the search will guarantee that no components will be searched unnecessarily.

3.2.2 Reach-Based Routing [14]

Let $u, v, w \in V$ and P a shortest u-w path containing v. Define

$$r_{v,P} := \min\{d_{u,v}, d_{v,w}\} \tag{3.2.1}$$

as the reach of v with respect to P. Define $\mathcal{P}_v$ as the set of all possible shortest paths between all possible u-w pairs that contain v. The reach of v is then defined as

$$r_v := \max\{r_{v,P} \mid P \in \mathcal{P}_v\}. \tag{3.2.2}$$

The reach of a node gives a sense of how far other nodes on shortest paths containing v can be.

When a shortest s-t path has to be computed, again, a bidirectional Dijkstra search is performed. If, however,

$$d_{s,v} > r_v \text{ and } d_{v,t} > r_v, \tag{3.2.3}$$

then there does not exist a shortest s-t path containing v and the search can be pruned at v. These conditions can also be checked using lower bounds on the mentioned distances. Suppose the forward search is about to scan node $v \in V$ and the backward search has not already scanned this node. In this case only $d_{s,v}$ is known. The distance $d_{v,t}$ can now be lower bounded by $d_{u,t}$, where u is the node that was last added to $\overleftarrow{\mathcal{T}}$. If both $d_{s,v}$ and $d_{u,t}$ are larger than the reach of v, the search can be pruned at v. Similar arguments hold for the backward search.

During preprocessing, an APSP computation can be performed to compute the reach of all nodes. However, this is very costly and can be easily improved upon. For example, one could compute only partial shortest path trees, with upper bounds on the reach as a result. Condition (3.2.3) for pruning at v still suffices if r_v represents an upper bound on the reach of v.

3.2.3 Graph Separators

The speed-up techniques in this category exploit the separability of planar graphs [21]. Most graphs, for which shortest paths need to be found, are not strictly planar. However, if they only have a small set of non-planar edges, it is still possible to find a small set of separators. These separators can either be edges or vertices. After removing them from the graph, it will be decomposed into several components. These components can then be used during preprocessing to obtain information that will speed up the query search. Both vertex and arc separator techniques will be discussed.

Vertex Separators [17]

During preprocessing a (preferably small) vertex separator set $V^* \subset V$ is computed. This set separates the graph G into several components after it is removed from G. Let $K \subset V \setminus V^*$ be such a component. Then we define $V_K^* \subset V^*$ as the minimal set that disconnects K from the remainder of G after removing V_K^*. Suppose we want to find a path from $u \in K_1$ to $v \in K_2$, with $K_1 \neq K_2$. Every u-v path has to go through at least one node in $V_{K_1}^*$ and at least one node in $V_{K_2}^*$. See Figure 3.2 for an example.

From the separator set V^* a new overlay graph G^* is constructed. An edge between nodes $u, v \in V^*$ is added to E^*, if there does not exist a shortest u-v path in G that contains another vertex $w \in V^*$. The new edge (u, v) is assigned weight equal to the shortest u-v path distance in G. By doing this for all possible

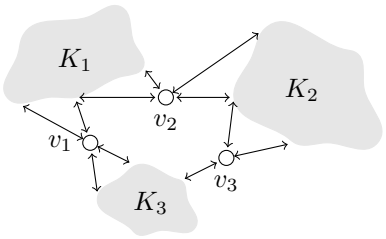


Figure 3.2: An example graph with separator vertices $V^* = \{v_1, v_2, v_3\}$ and components K_1, K_2 and K_3. Here $V^*_{K_1} = \{v_1, v_2\}$, $V^*_{K_2} = \{v_2, v_3\}$ and $V^*_{K_3} = \{v_1, v_3\}$.

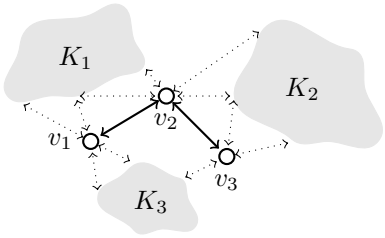


Figure 3.3: An example graph with overlay graph for the graph in Figure 3.2. Note the lack of edges between v_1 and v_3.

node pairs of V^*, an overlay graph is constructed that preserves shortest path distances. See Figure 3.3.

The bidirectional Dijkstra search, that is run during a query, can now be pruned, whenever it reaches one of the separator nodes. The search will then be proceeded from the corresponding node in the overlay graph.

Furthermore, this method can also be extended to a multilevel variant, where repeatedly a vertex separator set in the previous overlay graph is computed. This will result in shortcuts over larger distances that can result in faster query-times. However, it also requires more preprocessing time and more storage space.

The query-times can be reduced even further by adding more shortcut edges in the following way. For all vertices u in component K, the shortest paths to/from all $v \in V^*_K$ are computed and upward/downward shortcuts $(u, v)/(v, u)$ are added. Before using these upward edges to go to the overlay graph directly, one has to check if the source and target node are in the same component. If this is the case, then proceeding to the overlay graph can result in non-optimal paths. So, in such a case, a local bidirectional Dijkstra search can be performed instead. Note that adding these additional shortcuts requires even more preprocessing time and storage space.

Arc Separators

Similar to the Geometric Container and Arc Flag methods, the graph is split into k balanced cells, $\mathcal{K} := \{K_1, \ldots, K_k\}$, with (preferably) a low number of arcs between the cells. The arcs between the cells are now the separators, with the boundary nodes incident to these arcs.

Again, an overlay graph is constructed. This time, all separator arcs and boundary nodes are added. In addition, for every cell K_i and every pair of boundary nodes $b_1, b_2 \in K_i$ a shortcut edge (b_1, b_2) is added, with weight equal to the shortest b_1-b_2 path, restricted to the cell K_i. Thus, for every cell an overlay clique of all the boundary nodes is added. Once again, this method can be extended by computing a multilevel graph partitioning and adding several overlay graphs.

The query algorithm is similar to that of the vertex separator method. A

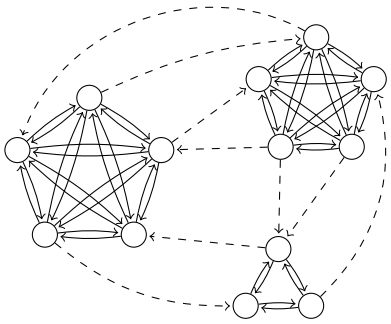


Figure 3.4: An example graph with dashed arc separators between three cells, with boundary node cliques.

bidirectional Dijkstra search is performed, but whenever a boundary node is reached, the search will be lifted to the overlay graph.

The Customisable Route Planning (CRP) algorithm [6] splits the preprocessing into two parts. In the first part, a multilevel partition of the graph is computed and the topology of the graph is captured. Note that this part is independent of the cost function, so it can be skipped if only the cost function of the graph is changed. The second part is dependent of the cost function and computes the cost of the shortcuts in the overlay cell cliques.

3.3 Table look-up techniques

Theoretically, the fastest query time can be achieved by performing an APSP computation and store all shortest paths in a table during preprocessing. An s-t query would then only require one table look-up. PHAST [7] is an algorithm that can solve an APSP computation on large graphs in reasonable time. However, storing the distances of all possible s-t pairs would require $O(V^2)$ space, which is prohibitive. The following techniques exploit the fast APSP computation of PHAST, while using less storage space to get a better trade-off.

3.3.1 Compressed Path Databases (CPD)

In this robust speed-up technique a PHAST computation is performed in the preprocessing phase. This way the shortest path for every possible pair of nodes is known. The first edge of these paths is stored. During the query phase, where the shortest path between $s \in V$ and $t \in V$ has to be found, one can recursively call the first edge of the remaining shortest path to t, since all those edges are stored during preprocessing. This query algorithm is very fast, because it only has to visit the nodes that are on the shortest s-t path. However, despite sophisticated data compression methods, the storage space that is needed to store the first edge of every possible path is still very large.

3.3.2 Hub Labelling (HL) [3]

During the preprocessing phase of HL, every node $u \in V$ is assigned two labels, $\vec{L}_u$ and $\overleftarrow{L}_u$, that store a set of vertices/hubs $\vec{H}_u, \overleftarrow{H}_u \subset V$ and the distances $d_{u,v}$ and $d_{v,u}$, for all $v \in \vec{H}_u$ and $v \in \overleftarrow{H}_u$, respectively. These labels are chosen in such a way that, for every pair $u, w \in V$, the shortest u-w path contains at least one node v such that $v \in \vec{L}_u \cap \overleftarrow{L}_w$. The query algorithm, then finds the shortest s-t path by

$$d_{s,t} = \min_{u \in \vec{L}_s \cap \overleftarrow{L}_t} d_{s,u} + d_{u,t} \tag{3.3.1}$$

One way to find such a labelling is by using an CH ranking. The hubs of a node v then include all the nodes with equal or higher rank than v.

3.3.3 Transit Node Routing (TNR) [1]

TNR selects a subset $T \subset V$ of transit nodes, for which an APSP computation is performed, using PHAST. The result is stored in a distance table. For every vertex $u \in V \setminus T$ a set of access nodes $\vec{A}_u \subset T$ is computed, which contain all $v \in T$, that are the first transit nodes on a shortest path starting in u. All the distances from u to the access nodes are stored as well. Analogous distances from $\overleftarrow{A}_u$ are stored, for all $u \in V$. The shortest path is then found by minimizing the cost of the s-a_s-a_t-t path, where $a_s \in \vec{A}_s$ and $a_t \in \overleftarrow{A}_t$. It is however possible that the shortest s-t path does not contain a transit node. Since this would result in a incorrect shortest path, the query is first classified as local or non-local. Here, local means that it could be possible that there are no transit nodes on the shortest s-t path. An alternative algorithm is needed to solve these local queries.

Choosing the transit nodes

Both vertex separators and the boundary nodes of an arc separator are a logical choice for the transit node set, since, in these cases, locality classification is straightforward. Even though TNR, with vertex separators as transit nodes, seems very similar to the extended vertex separator technique, it has a different trade-off. In both cases the paths from vertices to its separator set and the paths between all separator nodes is precomputed. However, the separator technique adds shortcuts to an overlay graph, that has to be searched in query-time, whereas TNR stores all computed paths in tables. Hence, TNR will have faster query-times, but requires more storage space.

The transit node set can also be constructed using the 'important' nodes of a hierarchy based technique, e.g. CH or HH, as these nodes will, in general, be visited more often. The locality classification is in this case less straightforward.

3.4 Combining speed-up techniques

Speed-up techniques that exploit different features of the graph can be easily combined to get even better performance. The following are two examples of speed-up techniques that are well suited to combine.

3.4.1 Shortcuts + Reach + ALT

The strength of reach-based routing lays in pruning vertices that meet condition (3.2.3), so the search will skip nodes that will not lead to a shortest path. Adding shortcuts to a graph, before computing the reach, will make this method more effective. Methods that add shortcuts, e.g. CH and HH, will reduce the number of vertices on long distance paths. The reach of the vertices, on this long distance path, is then reduced, as is shown in Example 3.4.1. The lower reach will have a positive effect on the performance, since condition (3.2.3) is met more often, resulting in searches that are pruned more often.

Example 3.4.1. Consider the graph in Figure 3.5. The weight of the arcs for both directions is stated above the arcs. The corresponding reach is stated below the vertices. After adding a shortcut for both directions, the reach of intermediate nodes is reduces significantly, as is shown in Figure 3.6.

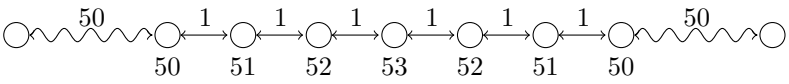


Figure 3.5: An example graph. The reach is stated below the vertices.

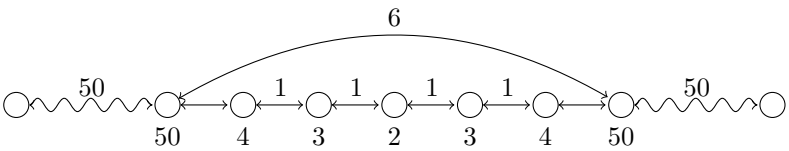


Figure 3.6: An example graph with shortcuts. The reach is stated below the vertices.

After computing the reach on the graph with shortcuts, one can apply ALT efficiently on a small subset of the graph. Applying ALT only on the vertices with high reach will reduces the number of landmark distances that have to be precomputed significantly. Furthermore, the lower bounds that are computed for ALT can be directly applied to the reach conditions, with an even better performance as result. Alternatively, ALT can be applied to a core graph, like the overlay graph constructed by HH, since the landmark distances and bounds are not affected by the shortcuts.

3.4.2 TNR + Arc Flags

The bottleneck of the TNR query algorithm, for an arbitrary s-t pair, is comparing the transit node sets $\vec{A}_s$ and $\overleftarrow{A}_t$ to find the shortest path. The query time can be improved by using arc flags to decrease the number of table look-ups. The transit node set T is divided into k disjoint cells $\mathcal{K} := \{K_1, \ldots, K_k\}$, with $K_i \subset T$, for all $i \in \{1, \ldots, k\}$. For every $s \in V$ and every $u \in \vec{A}_s$, k bits are stored. The i-th bit of u is set if and only if there exists a node $v \in K_i$ such that

$$d_{s,u} + d_{u,v} = \min_{w \in \vec{A}_s} \{d_{s,w} + d_{w,v}\}. \tag{3.4.1}$$

Analogous bits are set for the backward transit node sets. During an s-t query, only the $u \in \vec{A}_s$ and $v \in \overleftarrow{A}_t$, for which the bits of the corresponding cells are set, have to be compared.

3.5 Discussion

The algorithms, discussed in this chapter, preprocess the graph and compute, during a query, the distance of the shortest path, using the preprocessed information. In practice, however, this can be considered insufficient. This section will cover several topics that could improve the practical use of the algorithms.

3.5.1 Path Construction

The query algorithms, that are discussed so far, only return the distance of the requested shortest path. Outputting the actual shortest path can be desirable, but is for some techniques non-trivial. How this path can be constructed depends on the preprocessed information that is used.

Predecessors

Methods that do not skip any nodes during the query algorithm, like most goal directed methodes, can store predecessor nodes, as is done in the original Dijkstra algorithm. The path can then be constructed by recursively retrieving all predecessor nodes, starting with the target node.

Small Shortcuts

Constructing the path using predecessors in a graph with shortcuts can result in a partial path, since the shortcuts represent paths in the original graph. Methods that add shortcuts, that only bypass one node per shortcut, can store the corresponding intermediate node, together with the shortcut. Whenever the path needs to be constructed, one can retrieve the intermediate node. CH is an example of an algorithm that uses such shortcuts. Shortcuts on high levels may skip a node that also was connected with shortcuts. This way a shortcut can span across multiple nodes from the original graph. In this case a path needs to be constructed by recursively unpacking the underlying shortcuts.

Large Shortcuts

A path containing shortcuts that span across multiple nodes, in algorithm such as HH and CRP, can be constructed by either storing all intermediate nodes, or performing a local bidirectional search to find intermediate nodes. The former results in more storage space, whereas the latter requires longer query time.

3.5.2 Dynamic Graphs

A drawback of the discussed techniques is the staticity of graphs that is mandatory to perform the query algorithms. In practice, however, edge weights can change frequently. Whenever the graph changes, the previous preprocessed

information could be flawed and the graph has to be preprocessed again. Re-preprocessing the entire graph is costly and can often be prevented.

- In some cases it is possible to do only a partial preprocessing phase. For instance, CH can maintain the previous ordering of importance and recalculate only those shortcuts that could be affected by the changes to the graph.
- Another approach is to split the entire preprocessing phase into a weight-dependent and a weight-independent phase. CRP uses this approach [6] by making a weight-independent partitioning and overlay, that can be maintained on weight changes. Weight-independent contraction orders, in hierarchy-based techniques, can also be used to split the preprocessing phase.
- Under special circumstances it is possible to run the algorithm with unchanged preprocessing information and still get correct results. ALT, for example, will keep correct bounds if the weight on edges is only increased.

3.5.3 Multi commodity

The query algorithms of the discussed speed-up techniques focus on solving P2P instances. However, it can be desirable to compute the shortest path distance for more than one O-D pair. Straightforwardly, all O-D pair queries can be solved separately. Using hierarchical methods in combination with buckets [19] or a restricted implementation of PHAST [5], one can improve upon solving a P2P for every O-D pair separately.

In the special case of an *one-to-all*[5] shortest path problem, one could argue that the original Dijkstra algorithm is optimal, since it constructs a shortest path tree, while visiting the minimal amount of vertices. However, the PHAST algorithm achieves better performance by exploiting the ability to parallelize calculations [7].

[5]Recall that a one-to-all shortest path problem requires the shortest path from one source node to all other vertices of the graph.

4. Flows

4.1 Traffic assignment problem

A network has often more than one user travelling between different origins and destinations. If many users travel over the same route simultaneously, congestion can occur. The above can be captured by a flow as was described in Chapter 1.

Recall Definition 1.0.4. A flow f is an SO for instance $G_{\mathcal{C},c}$ if it solves

$$\begin{aligned} &\text{minimize} && z(f) \\ &\text{subject to} && \sum_{P\in\mathcal{P}_i} f_P = r_i, \quad i \in \{1,\dots,k\}, \\ & && f_P \geq 0, \qquad\quad P \in \mathcal{P}. \end{aligned} \tag{4.1.1}$$

Assumption 4.1.1. In addition to the assumptions that, for all $a \in A$, the function $f_a \mapsto c_a(f_a)$ is continuous, non-negative and non-decreasing, we assume that, for all $a \in A$, c_a is continuously differentiable and

$$\frac{\partial^2 c_a(f_a)}{\partial f_a^2} \geq 0. \tag{4.1.2}$$

Lemma 4.1.2. *The objective function $z(f)$ of (4.1.1) is convex under Assumption 4.1.1.*

Proof. It suffices to show that the Hessian of $z(f)$ is positive definite. We have, for all $b \in A$,

$$\frac{\partial z(f)}{\partial f_b} = \frac{\partial}{\partial f_b}\sum_{a\in A} f_a c_a(f_a) \tag{4.1.3}$$

$$= c_b(f_b) + f_b\frac{\mathrm{d}c_b(f_b)}{\mathrm{d}f_b} \tag{4.1.4}$$

and, for all $a, b \in A$,

$$\frac{\partial^2 z(f)}{\partial f_b\,\partial f_a} = \begin{cases} 2\frac{\mathrm{d}c_a(f_a)}{\mathrm{d}f_a} + f_a\frac{\mathrm{d}^2 c_a(f_a)}{\mathrm{d}f_a^2}, & \text{if } a = b, \\ 0, & \text{otherwise.} \end{cases} \tag{4.1.5}$$

So, the Hessian is a diagonal matrix with all positive terms, implying that it is positive definite. Hence, $z(f)$ is convex. □

Remark 4.1.3. Under Assumption 4.1.1, SO program (4.1.1) has a unique minimum, since

- The feasible region of the SO program (4.1.1) is convex. It is easy to see that the linear equality constraints imply a convex feasible region. The non-negativity constraints will not alter this fact.
- The objective function $z(f)$ is convex by Lemma 4.1.2.

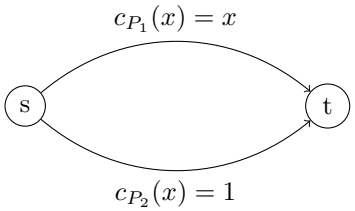


Figure 4.1: (Pigou's example) In this example there is one O-D pair with flow rate 1. The edges represent two different s-t paths, P_1 and P_2. Selfish routing results in a UE where all users take the upper path P_1, with total weight 1. A simple calculation shows that minimizing the total weight is achieved by evenly splitting the flow among the two paths, which results in total weight $\frac{3}{4}$.

In many cases users in a network tend to be selfish and will choose a path that has minimum weight. This selfish routing leads to the following definition of the user equilibrium:

Definition 4.1.4 (User equilibrium)**.** Let f be a feasible flow in a graph G with weight function c and flow rate r. Then f is called a *user equilibrium (UE)* if for all $i \in \{1, ..., k\}$ and $P, P^* \in \mathcal{P}_i$ with $f_P > 0$, we have

$$c_P(f) \leq c_{P^*}(f).$$

In other words, in a UE there does not exist flow that can be assigned to a path with less weight. Note that a UE and SO are not necessarily equal. See Figure 4.1 for an example. However, the two can be proven equivalent.

Proposition 4.1.5 (Equivalence of UE and SO)**.** *Let $G_{\mathcal{C},c}$ be an instance. Denote $\tilde{c}_a(x) := \frac{d}{dx}(x \cdot c_a(x))$ the* marginal weight function *and let $\tilde{c}_P(x) = \sum_{a \in P} \tilde{c}_a(x)$. Then f is the SO for $G_{\mathcal{C},c}$ if and only if f is a UE for $G_{\mathcal{C},\tilde{c}}$.*

In order to prove Proposition 4.1.5, consider the following lemma.

Lemma 4.1.6 (Characterization of the SO)**.** *Using the same notation as in Proposition 4.1.5, the following holds. Flow f is the SO for $G_{\mathcal{C},c}$ if and only if, for every $i \in \{1, \ldots, k\}$ and every pair $P, P^* \in \mathcal{P}_i$ with $f_P > 0$,*

$$\tilde{c}_P(f) \leq \tilde{c}_{P^*}(f).$$

Proof. This proof is based on a proof described in [31]. The necessary conditions for a flow f to be an SO and therefore the minimum of linear program (4.1.1), are given by the first-order conditions for a stationary point of the Lagrangian program

$$\begin{aligned} &\text{minimize} \quad && L(f,u) = z(f) + \sum_{i=1}^{k} u_i \left(r_i - \sum_{P \in \mathcal{P}_i} f_P \right), \\ &\text{subject to} \quad && f_P \geq 0, \quad P \in \mathcal{P}, \end{aligned} \tag{4.1.6}$$

where the variables u_i are the Lagrange multipliers associated with the flow

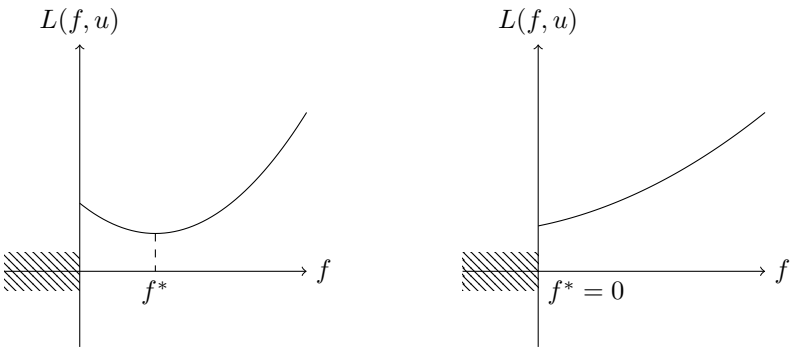


Figure 4.2: First-order conditions for Lagrangian with non-negative flow constraints. Left: Internal minimum with $f^* \geq 0$ and $\frac{\partial L(f^*,u)}{\partial f_P} = 0$. Right: Constrained minimum with $f^* = 0$ and $\frac{\partial L(f^*,u)}{\partial f_P} \geq 0$.

conservation constraint of O-D pair (s_i, t_i). These necessary conditions are

$$f_P \frac{\partial L(f,u)}{\partial f_P} = 0, \qquad \forall P \in \mathcal{P}, \tag{4.1.7}$$

$$\frac{\partial L(f,u)}{\partial f_P} \geq 0, \qquad \forall P \in \mathcal{P}, \tag{4.1.8}$$

$$\frac{\partial L(f,u)}{\partial u_i} = 0, \qquad \forall i \in \{1,\ldots,k\}, \tag{4.1.9}$$

$$f_P \geq 0, \qquad \forall P \in \mathcal{P}. \tag{4.1.10}$$

Conditions (4.1.9) and (4.1.10) simply restate the flow conservation and non-negativity constraints of (4.1.1). Conditions (4.1.7), (4.1.8) and (4.1.10) are necessary to minimize the Lagrangian with the non-negativity constraint on the flow variable, see Figure 4.2. To simplify conditions (4.1.7) and (4.1.8) the derivatives are written as the sum of two terms as follows:

$$\frac{\partial L(f,u)}{\partial f_P} = \frac{\partial z(f)}{\partial f_P} + \frac{\partial}{\partial f_P}\left[\sum_{i=1}^{k} u_i \left(r_i - \sum_{\hat{P}\in\mathcal{P}_i} f_{\hat{P}}\right)\right]. \tag{4.1.11}$$

For $P \in \mathcal{P}_j$ we have that the second term is equal to $-u_j$. The derivative in the first term is given by

$$\begin{aligned}
\frac{\partial z(f)}{\partial f_P} &= \sum_{a\in A} \frac{\partial z(f)}{\partial f_a} \cdot \frac{\partial f_a}{\partial f_P} \\
&= \sum_{a\in A} \frac{\partial \left(\sum_{b\in A} c_b(f_b) f_b\right)}{\partial f_a} \cdot \delta_P^a \\
&= \sum_{a\in A} \delta_P^a \cdot \tilde{c}_a(f_a) \\
&= \tilde{c}_P(f).
\end{aligned}$$

So the first-order conditions for a stationary point of the Lagrangian program

can be written as

$$f_P\left(\tilde{c}_P(f) - u_i\right) = 0, \qquad \forall P \in \mathcal{P}_i, \forall i \in \{1, \dots, k\}, \tag{4.1.12}$$

$$\left(\tilde{c}_P(f) - u_i\right) \geq 0, \qquad \forall P \in \mathcal{P}_i, \forall i \in \{1, \dots, k\}, \tag{4.1.13}$$

$$\sum_{P \in \mathcal{P}_i} f_P = r_i, \qquad \forall i \in \{1, \dots, k\}, \tag{4.1.14}$$

$$f_P \geq 0, \qquad \forall P \in \mathcal{P}. \tag{4.1.15}$$

So, for a path P in commodity i with $f_P > 0$ equation (4.1.12) gives that $\tilde{c}_P(f) = u_i$. Inequality (4.1.13) then gives that every other path $P^* \in \mathcal{P}_i$ has the property

$$\tilde{c}_{P^*}(f) \geq u_i = \tilde{c}_P(f).$$

So the fact that f is an SO flow implies that, for every $i \in \{1, \dots, k\}$ and every pair $P, P^* \in \mathcal{P}_i$ with $f_P > 0$,

$$\tilde{c}_P(f) \leq \tilde{c}_{P^*}(f).$$

To proof the reversed implication, it is sufficient that program (4.1.6) has an unique solution, see Remark 4.1.3. This concludes the proof.

□

With Lemma 4.1.6 the proof of Proposition 4.1.5 is trivial.

Proof of Proposition 4.1.5. The correctness of this statement follows directly from Lemma 4.1.6 and Definition 4.1.4. □

Consider Pigou's example in Figure 4.1. Using the marginal weight functions, as defined in Proposition 4.1.5, results in $\tilde{c}_{P_1}(x) = 2x$ and $\tilde{c}_{P_2}(x) = 1$. Some thought reveals that the flow for a UE with respect to $\tilde{c}$, as Proposition 4.1.5 indicates, indeed splits the flow evenly among the two paths.

4.2 Computation of the UE

Consider an instance $G_{\mathcal{C},c}$ for which we want to find a UE flow f. To use Proposition 4.1.5 we need to find a function, say ϕ, for which the marginal weight function is equal to c. So, let $\phi_a(x) = \int_0^x c_a(y)\,\mathrm{d}y \cdot x^{-1}$ and define the *potential function*

$$\Phi(f) := \sum_{a \in A} \int_0^{f_a} c_a(x)\,\mathrm{d}x. \tag{4.2.1}$$

Note that Lemma 4.1.2 still holds for $\Phi(f)$.

Corollary 4.2.1. *Let $G_{\mathcal{C},c}$ be an instance. Then a feasible flow f is a UE for $G_{\mathcal{C},c}$ if and only if f is the SO for $G_{\mathcal{C},\phi}$.*

Proof. Follows directly from Proposition 4.1.5. □

Corollary 4.2.1 states that finding a UE for an instance $G_{\mathcal{C},c}$ is equal to finding a solution to the following minimisation problem:

$$\begin{aligned} \text{minimize} \quad & \Phi(f) = \sum_{a \in A} \int_0^{f_a} c_a(x)\,\mathrm{d}x \\ \text{subject to} \quad & \sum_{P \in \mathcal{P}_i} f_P = r_i, && i \in \{1, \ldots, k\}, \\ & f_P \geq 0, && P \in \mathcal{P}_i,\ i \in \{1, \ldots, k\}. \end{aligned} \tag{4.2.2}$$

One way to solve this linear program is by the Frank-Wolfe algorithm [11]. This algorithm can be summarized as follows:

1. *Initialisation.* Find a feasible solution f^0 for (4.2.2) and let $n = 0$.
2. *Direction finding.* Find g^n that solves

$$\begin{aligned} \text{minimize} \quad & \nabla\Phi\left(f^n\right)\left(g^n - f^n\right)^T \\ \text{subject to} \quad & \sum_{P \in \mathcal{P}_i} g_P^n = r_i, && i \in \{1, \ldots, k\}, \\ & g_P^n \geq 0, && P \in \mathcal{P}_i,\ i \in \{1, \ldots, k\}. \end{aligned} \tag{4.2.3}$$

3. *Step-size determination.* Find α^n that solves

$$\min_{0 \leq \alpha^n \leq 1} \Phi\left[f^n + \alpha^n\left(g^n - f^n\right)\right]. \tag{4.2.4}$$

4. *Move.* Let $f^{n+1} := f^n + \alpha^n\left(g^n - f^n\right)$.
5. *Convergence test.* If $\Phi\left(f^n\right) - \Phi\left(f^{n+1}\right) < \kappa$, stop[1]. Else, let $n := n + 1$ and repeat from step 2.

Similarly to the Primal-Dual Algorithm, a new linear program has to be solved in every iteration. Solving (4.2.3) will turn out to be easy when applying the Frank-Wolfe algorithm to the traffic assignment problem. A better understanding of the structure of program (4.2.3) can be obtained by taking a closer look at the objective function. The objective function of (4.2.3) can be further written as

$$\begin{aligned} \nabla\Phi(f^n)(g^n - f^n)^T &= \sum_{a \in A} \left(\frac{\partial\Phi\left(f^n\right)}{\partial f_a^n}\right)\left(g_a^n - f_a^n\right) && (4.2.5) \\ &= \sum_{a \in A} c_a\left(f_a^n\right)\left(g_a^n - f_a^n\right) && (4.2.6) \\ &= \sum_{a \in A} c_a\left(f^n\right) g_a^n - \sum_{a \in A} c_a\left(f^n\right) f_a^n. && (4.2.7) \end{aligned}$$

The first sum of (4.2.7) corresponds to the total weight of flow g^n through a graph with constant weight functions $c_a\left(f^n\right)$, whereas the second sum is just a constant. To minimize this total weight, it is sufficient to calculate the shortest path for every O-D pair in the graph with constant edge weight functions

[1] Any other convergence criteria could be used.

$c_a(f^n)$. As a result, the UE can be found by an iterative process of shortest path calculations.

Note that the weight functions change with every iteration. Hence, one has to consider a shortest path algorithm that is suited for dynamic graphs, such as the Bidirectional Dijkstra algorithm or the Primal-Dual Algorithm.

Bibliography

[1] Holger Bast et al. "In Transit to Constant Time Shortest-path Queries in Road Networks". In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 46–59.

[2] Richard Bellman. "On a Routing Problem". In: *Quarterly of Applied Mathematics* 16.1 (1958), pp. 87–90. ISSN: 0033569X, 15524485.

[3] Edith Cohen et al. "Reachability and Distance Queries via 2-hop Labels". In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 937–946. ISBN: 0-89871-513-X.

[4] George B. Dantzig. "On the Shortest Route Through a Network". In: *Manage. Sci.* 6.2 (Jan. 1960), pp. 187–190. ISSN: 0025-1909. DOI: 10.1287/mnsc.6.2.187. URL: http://dx.doi.org/10.1287/mnsc.6.2.187.

[5] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Faster Batched Shortest Paths in Road Networks". In: *11th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*. Ed. by Alberto Caprara and Spyros Kontogiannis. Vol. 20. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 52–63. ISBN: 978-3-939897-33-0. DOI: 10.4230/OASIcs.ATMOS.2011.52. URL: http://drops.dagstuhl.de/opus/volltexte/2011/3266.

[6] Daniel Delling et al. "Customizable Route Planning". In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Springer Verlag, May 2011. URL: https://www.microsoft.com/en-us/research/publication/customizable-route-planning/.

[7] Daniel Delling et al. *PHAST: Hardware-Accelerated Shortest Path Trees*. Tech. rep. Sept. 2010. URL: https://www.microsoft.com/en-us/research/publication/phast-hardware-accelerated-shortest-path-trees/.

[8] E. W. Dijkstra. "A Note on Two Problems in Connexion with Graphs". In: *Numer. Math.* 1.1 (Dec. 1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390.

[9] Jittat Fakcharoenphol and Satish Rao. "Planar Graphs, Negative Weight Edges, Shortest Paths, and Near Linear Time". In: *J. Comput. Syst. Sci.* 72.5 (Aug. 2006), pp. 868–889. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2005.05.007. URL: http://dx.doi.org/10.1016/j.jcss.2005.05.007.

[10] L. R. Ford. *Network Flow Theory*. P-923. California, Santa Monica, Aug. 1956, pp. 87–90.

[11] Marguerite Frank and Philip Wolfe. "An algorithm for quadratic programming". In: *Naval Research Logistics Quarterly* 3.1-2 (1956), pp. 95–110. ISSN: 1931-9193.

[12] Robert Geisberger et al. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks”. In: *Proceedings of the 7th International Conference on Experimental Algorithms*. WEA'08. Provincetown, MA, USA: Springer-Verlag, 2008, pp. 319–333. ISBN: 3-540-68548-0, 978-3-540-68548-7.

[13] Andrew V. Goldberg and Chris Harrelson. “Computing the Shortest Path: A Search Meets Graph Theory”. In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '05. Vancouver, British Columbia: Society for Industrial and Applied Mathematics, 2005, pp. 156–165. ISBN: 0-89871-585-7.

[14] Ron Gutman. “Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks.” In: Jan. 2004, pp. 100–111.

[15] Richard V. Helgason, Jeffery L. Kennington, and B. Douglas Stewart. “The One-to-one Shortest-path Problem: An Empirical Analysis with the Two-tree Dijkstra Algorithm”. In: *Computational Optimization and Applications* 2.1 (June 1993), pp. 47–75. ISSN: 0926-6003. DOI: 10.1007/BF01299142. URL: http://dx.doi.org/10.1007/BF01299142.

[16] Moritz Hilger et al. “Fast Point-to-Point Shortest Path Computations with Arc-Flags”. In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. American Mathematical Society, 2009, pp. 41–72.

[17] Martin Holzer, Frank Schulz, and Dorothea Wagner. “Engineering Multilevel Overlay Graphs for Shortest-path Queries”. In: *J. Exp. Algorithmics* 13 (Feb. 2009), 5:2.5–5:2.26. ISSN: 1084-6654.

[18] Lasse Kliemann and Peter Sanders, eds. *Algorithm Engineering: Selected Results and Surveys*. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-49487-6.

[19] Sebastian Knopp et al. “Computing Many-to-many Shortest Paths Using Highway Hierarchies”. In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. New Orleans, Louisiana: Society for Industrial and Applied Mathematics, 2007, pp. 36–45.

[20] Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, eds. *Algorithmics of Large and Complex Networks: Design, Analysis, and Simulation*. Berlin, Heidelberg: Springer-Verlag, 2009. ISBN: 978-3-642-02093-3.

[21] Richard J. Lipton and Robert E Tarjan. *A Separator Theorem for Planar Graphs*. Tech. rep. Stanford, CA, USA, 1977.

[22] Jens Maue, Peter Sanders, and Domagoj Matijevic. “Goal-directed Shortest-path Queries Using Precomputed Cluster Distances”. In: *J. Exp. Algorithmics* 14 (Jan. 2010), 2:3.2–2:3.27. ISSN: 1084-6654. DOI: 10.1145/1498698.1564502.

[23] E.F. Moore. *The Shortest Path Through a Maze*. Vol. 3523. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959.

[24] T. A. J. Nicholson. “Finding the Shortest Route between Two Points in a Network”. In: *The Computer Journal* 9.3 (1966), pp. 275–280.

[25] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. (Originally published: Englewood Cliffs, NJ : Prentice Hall, Inc., 1982). Mineola, NY, USA: Dover Publications, Inc., 1998. ISBN: 0-486-40258-4.

[26] Ira Sheldon Pohl. "Bi-directional and Heuristic Search in Path Problems". PhD thesis. Stanford, CA, USA, 1969.

[27] Tim Roughgarden. "Routing Games". In: *Algorithmic Game Theory*. Ed. by Noam Nisan et al. New York, NY, USA: Cambridge University Press, 2007. Chap. 18, pp. 461–486. ISBN: 0-521-87282-0.

[28] Peter Sanders and Dominik Schultes. "Highway Hierarchies Hasten Exact Shortest Path Queries". In: *Proceedings of the 13th Annual European Conference on Algorithms*. ESA'05. Palma de Mallorca, Spain: Springer-Verlag, 2005, pp. 568–579. ISBN: 3-540-29118-0, 978-3-540-29118-3. DOI: 10.1007/11561071_51. URL: http://dx.doi.org/10.1007/11561071_51.

[29] Dominik Schultes and Peter Sanders. "Dynamic Highway-node Routing". In: *Proceedings of the 6th International Conference on Experimental Algorithms*. WEA'07. Rome, Italy: Springer-Verlag, 2007, pp. 66–79. ISBN: 978-3-540-72844-3.

[30] Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. "Using Multi-level Graphs for Timetable Information in Railway Systems". In: *Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments*. ALENEX '02. London, UK, UK: Springer-Verlag, 2002, pp. 43–59. ISBN: 3-540-43977-3.

[31] Yosef Sheffi. *Urban Transportation Networks*. Englewood Cliffs, NJ 07632: Prentice Hall, Inc., 1985. ISBN: 0-139-39729-9.

[32] Mikkel Thorup. "Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem". In: *J. Comput. Syst. Sci.* 69.3 (Nov. 2004), pp. 330–353. ISSN: 0022-0000. DOI: 10.1016/j.jcss.2004.04.003. URL: http://dx.doi.org/10.1016/j.jcss.2004.04.003.

[33] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. "Geometric Containers for Efficient Shortest-path Computation". In: *J. Exp. Algorithmics* 10 (Dec. 2005). ISSN: 1084-6654. DOI: 10.1145/1064546.1103378.

[34] Xugang Ye, Shih-Ping Han, and Anhua Lin. "A Note on the Connection Between the Primal-Dual and the A* Algorithm". In: *International Journal of Operations Research and Information Systems (IJORIS)* 1.1 (Jan. 2010), pp. 73–85.